Algorithms and Data Structures Portfolio

Domenico Di Ruocco,
CODE University of Applied Sciences,
SE02 Algorithms and Data Structures,
Spring Semester 2021.

Table of Contents

---

1. Introduction

In Computer Science, an algorithm is a set of instructions that must be followed in a fixed order to calculate an answer to a mathematical problem [Cambridge Dictionary]. Since it is common to find more than one algorithm that has been developed to solve the same problem, we need a way to analyze and compare them.

1.1. Time Complexity

One way to compare two algorithms that solve the same problem, assuming that the solutions provided by both are correct, is to compare the time it takes each of them to get to the solution. The problem with this method is that it depends on the hardware where the algorithm runs. It is for this reason that for machine-independent algorithm design we consider our algorithm to be running on a hypothetical machine called the "Random Access Machine" or RAM.

On the RAM, we consider each simple operation (+, *, -, =, if, call) and each memory access to take one time step, while loops and subroutines are considered to be the composition of many single-step operations.

1.2. Space Complexity

Another way to compare two algorithms is to compare the total space it takes them to get to the solution. The total space includes the size of the input and the auxiliary space, which is the extra or temporary space used by the algorithm.

1.3. Asymptotic Notation

We can use the RAM model to determine the number of steps it will take an algorithm to end with an input we choose, but estimating the worst, average, and best case runtime scenario with the RAM model can be inconvenint.

In [1]:
```
def even_numbers_avg(array):
    '''
    This function returns either the average of the sum of even numbers,
    or None.
    '''
    even_sum = 0                    #1 time step
    even_count = 0                  #1 time step
                                    #n times:
    for n in array:                     #1 time step
        if n % 2 == 0:                  #1 time step
            even_sum += n                   #1 time step
            even_count +=1                  #1 time step

    if even_count > 0:              #1 time step
        return even_sum / even_count      #1 time step
    else:                           #1 time step
        return None                     #1 time step
```

In the example above, we can try to generalize the time complexity of this algorithm by counting every step, and we will find that in the worst case its time complexity will be: $T(n) = 5n + 6$. Its space complexity will be the size of the array n, plus the two variables we initialize.

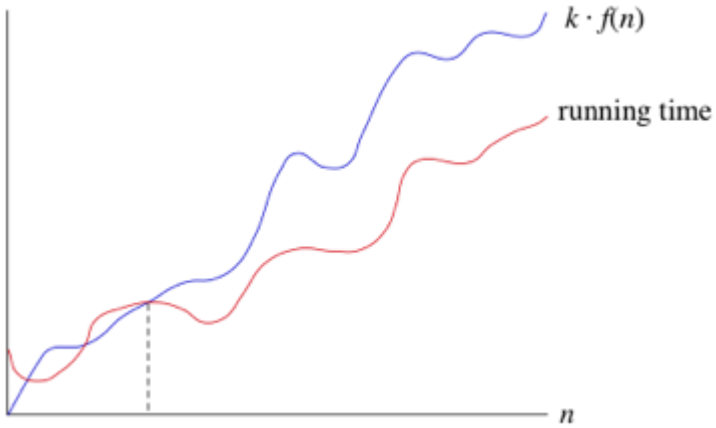
The problem with the notation we used above is that is difficult to work precisely with it. In the example above we can see that both return statements have been counted as well as every step in the for loop, which is correct for the worst but not for the average and best-case scenarios.

Since we can approximate an algorithm to a mathematical function, we can also determine its growth as a function of the input and define an upper bound function (Big O), a lower bound function (Big Ω), or both (Big Θ) in order to understand how it grows.

In Asymptotic Notation, we only consider the fastest-growing term without any multiplicative constant. E.g.: in the example above $T(n) = 5n + 6$ is $O(n)$ and not $O(5n)$.

1.3.1. Big O

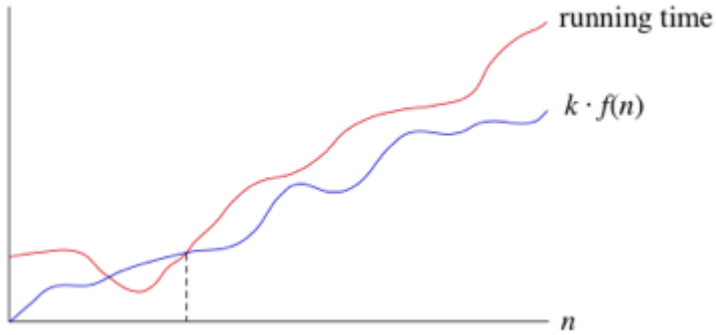
The Big O of a function is its asymptotic upper bound. This means that the running time of a function T will be always shorter than that of f. To generalize we can say that a function $T(n)$ is $O(f(n))$ if there is a constant k such that $T(n) < k \cdot f(n)$ for large enough n.



Big O. Image source: Khan Academy

1.3.2. Big Ω

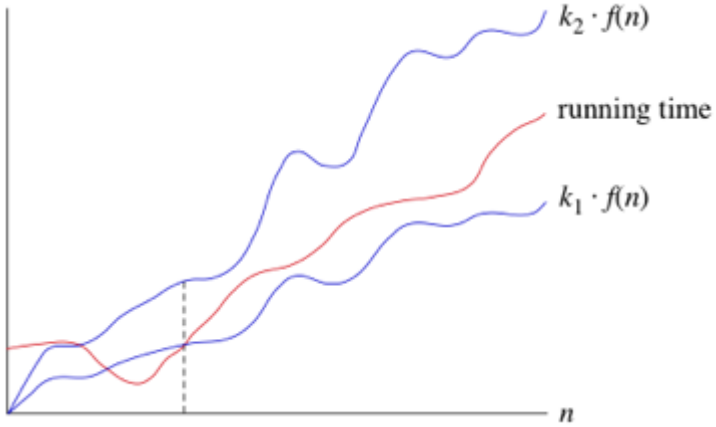
The Big Ω of a function is the asymptotic lower bound of that function. This means that the running time of a function T will always be longer than that of f. To generalize we can say that a function $T(n)$ is $\Omega(f(n))$ if there is a constant k such that $T(n) > k \cdot f(n)$ for large enough n.



Big Ω. Image source: Khan Academy

1.3.3. Big Θ

The Big Θ of a function is its asymptotic tight bound. This means that the function always runs in a time comprised between the run time of the two asymptotic bounds. To generalize we can say that a function $T(n)$ is $\Theta(f(n))$ if there are two constants k_1 and k_2 such that $T(n) \geq k_1 \cdot f(n)$ and $T(n) \leq k_2 \cdot f(n)$ for large enough n.



Big Θ. Image source: Khan Academy

1.3.4. Working with the Asymptotic Notation

1. Addition:

   We can sum two functions together and the result will be the dominant one:
   $f(n) + g(n) = \Theta(max(f(n), g(n)))$.

2. Multiplication:

   Multiplying a function by a constant will result in the constant being ignored. If instead, we are multiplying two functions, we proceed as follows:
   $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$.

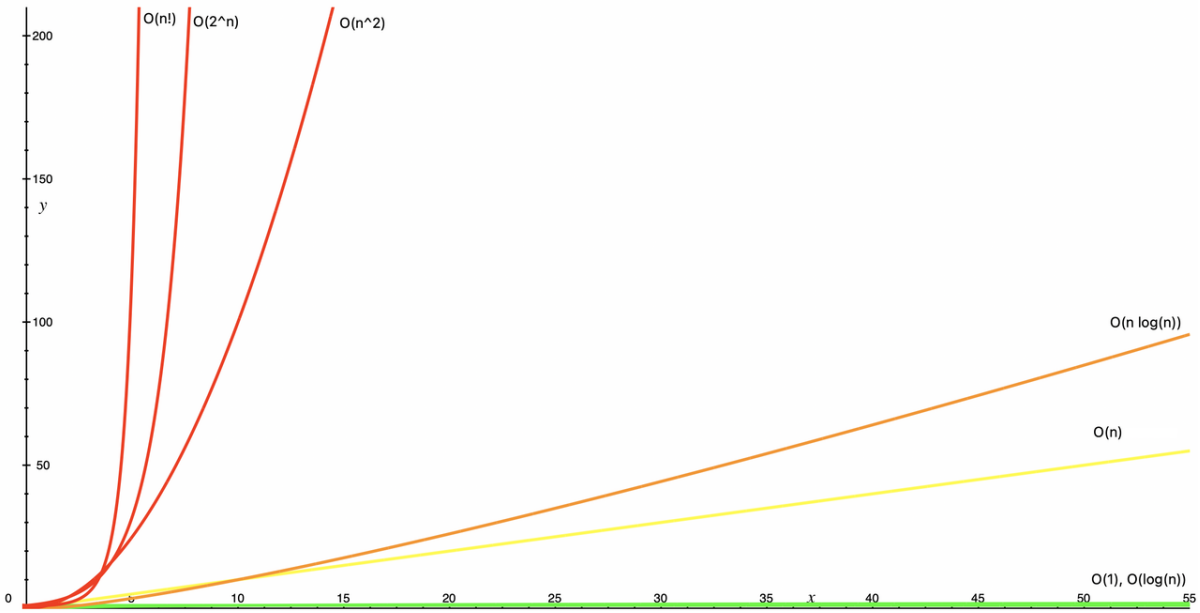
The same rules also apply to Big O and Big Ω

1.4. Order of Dominance in the Asymptotic Limit

Let's consider some common asymptotic growths, valid for both space and time complexity:

- Constant: $O(1)$
- Logarithmic: $O(log(n))$
- Linear: $O(n)$
- Quasilinear: $O(n \cdot log(n))$
- Quadratic: $O(n^2)$
- Exponential: $O(2^n)$
- Factorial: $O(n!)$

To understand the difference between some of the most common time complexities, take a look at the graph below, where the x-axis represents the size of the input of the functions and the y-axis represents the result of the functions.



We can see that the dominance order of these functions is:

$$n! >> 2^n >> n^2 >> n \cdot logn >> n >> log(n) >> 1$$

It is also clear that an efficient algorithm can really make the difference in terms of time and space efficiency, especially as the input size grows.

1.5. Algorithm Design Techniques

Another way to classify algorithms is based on the technique used to implement them. We will now examinate some common techniques and their pros and cons.

1.5.1. Brute Force Algorithms

Brute Force Algorithms are unefficient algorithms that instead of using other techniques to improve their efficiency, rely on computing power to try every possible combination. They are usually easy to implement, and the first solution that may come to mind when trying to solve a problem.
Examples of Brute Force Algorithms include the Selection Sort and Bubble Sort.

1.5.2. Divide and Conquer

Algorithms that use the divide and Conquer technique divide the problem they are trying to solve into sub-problems, recursively solve these, and then recombine them to get the final answer.

We can calculate the time efficiency of these algorithms, and any other recursive algorithm with the Master Theorem, using the following formula: $T(n) = aT(n/b) + f(n)$, where: n is the size of the input; a is the number of subproblems in the recursion; n/b is the size of each subproblem. All subproblems are assumed to have the same size; and $f(n)$ is the cost of the work done outside the recursive call.

One example of algorithm that uses the Divide and Conquer technique is the MergeSort.

1.5.3. Greedy Algorithms

Greedy Algorithms are used in optimization problems. They always make the 'local optimum' choice to optimize a given objective. They do not guarantee the optimal solution to the problem they are given.

An example of greedy algorithm is Dijkstra's Algorithm.

1.5.4. Dynamic Programming

Dynamic Programming can be considered an optimization of recursion. Whenever an algorithm that uses this technique calls a recursive function, it stores the returned value in a data structure, so that if later the same calculation will be needed again, the algorithm will just look up the result in the data structure. This can improve the time complexity of a probel from exponential to polynomial.

1.5.5. Backtracking Algorithms

Backtracking Algorithms are usually recursive algorithms that try different possibilities until they find the right one. If a piece of the solution computed by the algorithm does not lead to a full solution, these algorithms remove that piece of the solution and backtract to the point where there was another possible alternative.

An example of algorithm that uses backtracking is the Depth-First Search Algorithm.

---

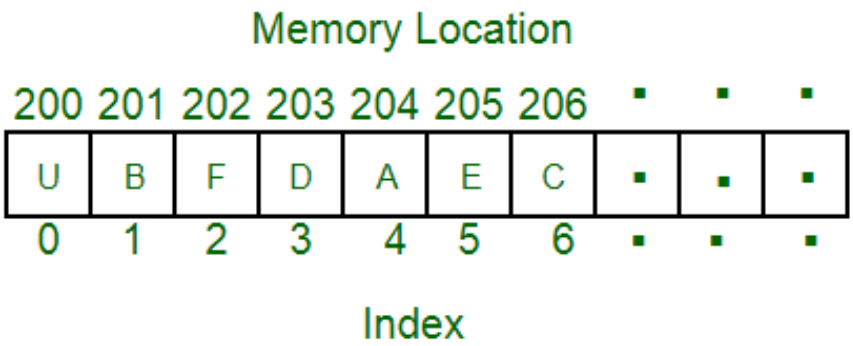
2. Data Structures

Data Structures are constructs that allow you to store and manage data values and provide you with methods to access or manipulate such data. There are different kinds of data structures available, each with its pros and cons. It is important to understand the strengths and weaknesses of different Data Structures in order to choose the right one for your use case and make your software more efficient.

Data structures can be classified into "contiguous" and "linked". The former are based upon arrays and the latter on pointers.

2.1. Arrays

An Array is an example of contiguous data structure. They are collections of data of fixed size allocated in contiguous memory locations, which make accessing the data values by index really efficient.



Array. Image source: Geeks for Geeks

Analysis of common operations on arrays:

- Access:

  Since the values are indexed and this is a data structure contiguous in memory it is possible to access data in the array with worst-case time complexity of $O(1)$ (constant time complexity);

- Search:

  The data stored in an array is not always sorted, so we need to assume that every memory slot could be searched before finding the element we are looking for, therefore the time complexity of this operation is $O(n)$ (linear time complexity);

- Insertion:

To insert an element at a specific index we might, in the worst case scenario, need to shift all the other elements in the array, so the time complexity for this operation will be $O(n)$;

- Deletion:

  To delete an element in an array we may run into the same issue of insertion, hence the time complexity will be, again, $O(n)$.

The space efficiency of arrays is also one of its major strengths since arrays are only made up of pure data, no space is wasted.

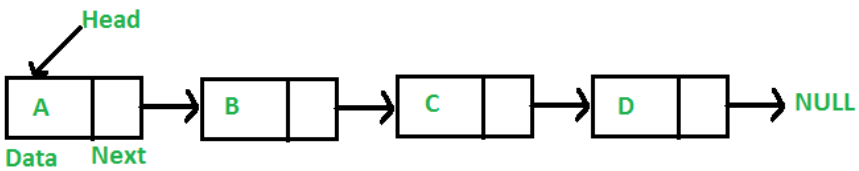
Another advantage of arrays is their memory locality, which takes full advantage of the speed of cache memory.

Their main disadvantage is the impossibility of changing their size while the program is running. It is however possible to avoid this limitation using dynamic arrays, arrays whose size doubles every time they are full.

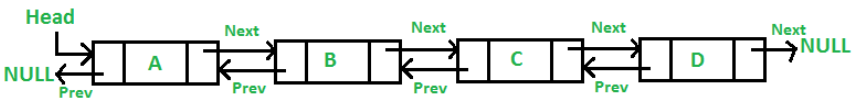
Arrays are very common data structures, and they can be used to store basically every kind of data. Other data structures, like Hash Tables, Graphs, and Heaps, are also based on them.

2.2. Linked Lists

Linked lists are an example of linked data structures. They are collections of data not allocated in contiguous memory locations but in which the elements are linked using a pointer to the next value in the case of a Singly Linked List or to both the previous and the next in the case of a Doubly Linked List. Linked lists are made up of "Nodes", the first one of which is called Head. Every node has a pointer that points to the next node (or to a null value in case it is the last element), while in case they are doubly linked list they also have a pointer to the previous value (or to a null value in case of the first element).



Singly Linked List. Image source: Geeks for Geeks



Doubly Linked List. Image source: Geeks for Geeks

Analysis of common operations on linked lists:

- Access / Search:

Linked Lists (both Singly and Dobly Linked) are not indexed data structures. This means that we cannot access one value directly, but we need to start from the Head (or also from the last element if we are in a Doubly Linked List) and follow the pointers until we find the element we are looking for or we find a null value. For this reason, the time complexity of this operation is $O(n)$;

- Insertion:

  To insert a new node "B" in a Singly Linked List, between two nodes "A" and "C", we just need to make sure that the node "A" points to node "B" and that node "B" points to node "C". If the list is a Doubly Linked one, we also need to make sure that the backward pointer of node "C" and "B" is set correctly. We can also add an element at the beginning of a Linked List by simply making it the new head and set its pointer to the previous head, and in case it is a Doubly Linked List we also need to point the backward pointer of the previous Head to the new Head. The time complexity of this operation, after you know the position where you want to insert a new element is $O(1)$, since you only need to change the value of the pointer(s);

- Deletion:

  The same concept from the insertion operation applies here, except that instead of changing the pointer(s) to include a new element we change them to exclude one. For this reason, the time complexity of this operation is also $O(1)$.

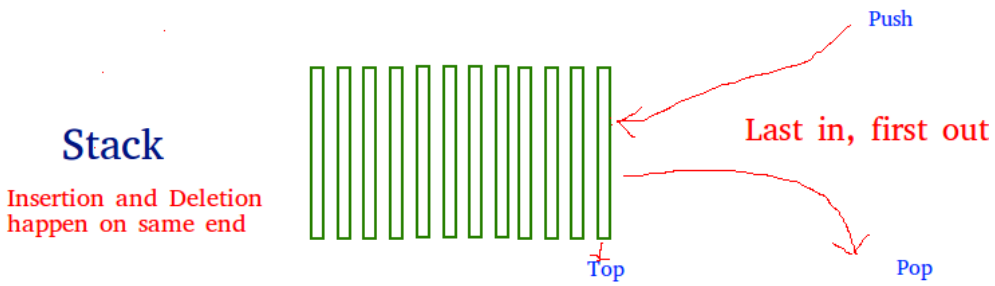
Linked lists are not really space-efficient since they need to store the pointers and the extra space needed will be $O(n)$.

As we have seen, their main disadvantages are that they are slow in searching and occupy more memory than arrays. Moreover, another disadvantage is that they don't benefit from the speed of cache memory since they are not stored contiguously.

Lists, just like arrays, are also very common data structures. They are mainly used where dynamic memory allocation is required. Trees are an example of data structure implemented using a modified version of linked lists.

2.3. Stacks

A Stack is a linear data structure. They allow two operations: insertion at the top (push) and read and removal at the top (pop). For this reason, stacks follow the "Last In First Out" or "LIFO" order.



Stacks. Image source: Geeks for Geeks

Analysis of common operations on Stacks:

- Access / Search:

  Access an element in a Stack entails that we need to read and remove the top item until we reach the one we're looking for or we are left with an empty Stack. Because of this, the time complexity of this operation will depend on its size and will therefore be $O(n)$;

- Insertion:

  We can only insert an element at the top of the Stack, and for this reason, this operation will have a time complexity of $O(1)$ (constant time);

- Deletion:

  Just like insertion, we can only delete the top element of a Stack and thus the time complexity of this operation will also be $O(1)$.

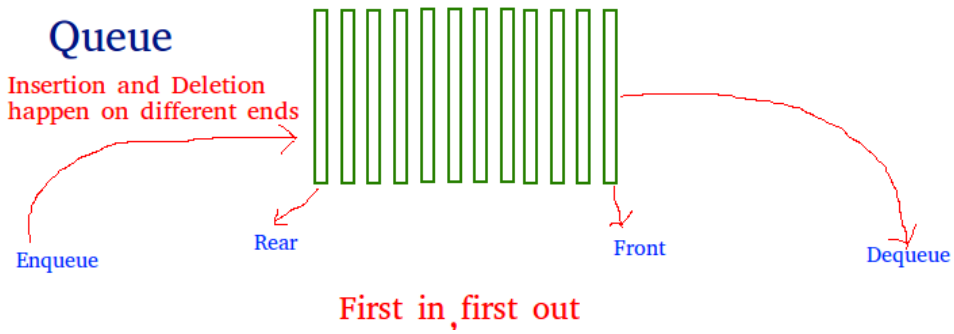
Depending on how Stacks are implemented they can be more or less space-efficient.

Their main advantage is that they are easy to implement and that that the insertion and deletion operations are really time efficient.

Uses of Stacks include situations in which the order in which the elements are inserted/deleted is not important, or when you specifically need to retrieve the last elements first. An example of implementation could be a social media feed or a chat app, because in both cases you need to retrieve the latest information first.

2.4. Queues

A Queue is a linear data structure similar to the Stack but that supports a different set of operations, Enqueue (inserting an item at the rear of the queue) and Dequeue (reading and deleting an item from the front of the queue). Instead of the LIFO, Queues follow the FIFO order (First In, First Out).



Queues. Image source: Geeks for Geeks

Analysis of common operations on Queues:

- Access / Search:

  Access an element in a Queue is similar to the access operation in a Stack, we need to read and remove (dequeue) the front item until we find the one we are looking for or no

elements are left in the Queue. The time complexity of this operation will hence be $O(n)$;

- Insertion:

  We can only insert an element at the rear of the Queue, and for this reason, this operation will have a time complexity of $O(1)$ (constant time);

- Deletion:

  We can only delete the front element of a Queue and thus the time complexity of this operation will also be $O(1)$.

The space complexity of stacks also depends on how they are implemented.

Just like Stacks, they are also easy to implement and that the insertion and deletion operations are really time efficient.

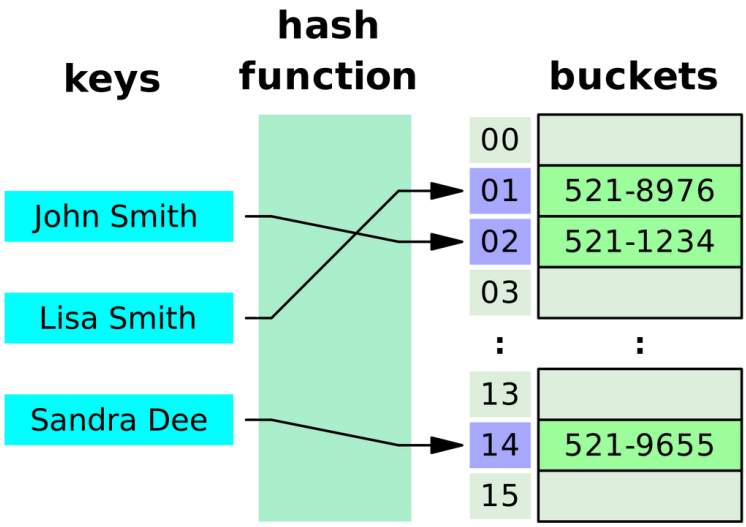
It is worth mentioning that another type of Queues, Priority Queues, are also widely used in computer science. They are different from a normal Queue because they don't follow the FIFO order, but they prioritize elements with a higher (or lower) priority. Priority Queues are usually implemented using the Heap Data Structure, so the time complexity to insert an element in a priority queue is the same of insertin an element in a Heap ($O(n * log(n))$). We will see them in action while analyzing the A* Algorithm.

Queues are useful in situations in which the order in which the elements are retrieved matters. An example of implementation is every service that needs to handle the users' requests in the order that they were made (like for online payments).

2.5. Hash Tables

A Hash Table is a data structure in which a key-value data pair is stored. It is usually implemented with an array and it works by hashing (generating a unique value, integer in this case) the key and storing the key-value data at the index returned by the hash function. In this way, given the key, it is really efficient to locate its value.

The main challenge of Hash Tables is to remain memory-efficient while avoiding collisions (having 2 or more elements at the same index).



Hash Table. Image source: Wikipedia

Analysis of common operations on Hash Tables:

- Search / Access:

  Since the data stored in a Hash Table is indexed, it will take constant time to search or access it ($O(1)$), even if depending on the hashing algorithm, it could depend on the size of the key. Other edge cases include collision, and we will take such situation into account later;

- Insertion:

  To insert an element in a Hash table, we just need to execute the hash function and insert the data in the array, which happens, depending on the hashing function, wither with a time complexity of $O(1)$ (constant time), or with time complexity dependent on the key size. This can change in case of collisions;

- Deletion:

  Deleting an element in a Hash Table is a process similar to searching for it, except that instead of reading it, it gets deleted. Normally this process happened with time complexity of $O(1)$, but also this operation can be slowed down by collisions.

Collisions are more frequent as the array fills up because the empty slots are fewer. It is common to use the variable $a = n/m$ (where n is the number of elements and m the length of the array) to measure how full is an array. Once a collision happens there are different ways to deal with it:

- Chaining:

  We add more than 1 key-value pair to the same index. In this way inserting a new element has always a time complexity of $O(1)$ while searching for an element a time complexity of $(O(1 + a))$;

- Open Addressing:

We store the elements in the same array without using additional data structures. Ways to find a new index include:

- Linear Probing:

  We store the colliding element in the first available slot in the array. This method, however, can be really inefficient. Inserting and searching for an element can have a time complexity of $O(n)$;

- Quadratic Probing:

  We find a new index by adding an arbitrary number that increases quadratically. Just like Linear probing, this method can be really inefficient with inserting and sorting operations that can have a time complexity of $O(n)$;

- Double Hashing:

  We generate a new hash if a collision is detected. We can choose between different functions to implement this technique, but generally, this method is more time-efficient compared to the other 2, especially in searching.

Hash Table are not really space-efficient.

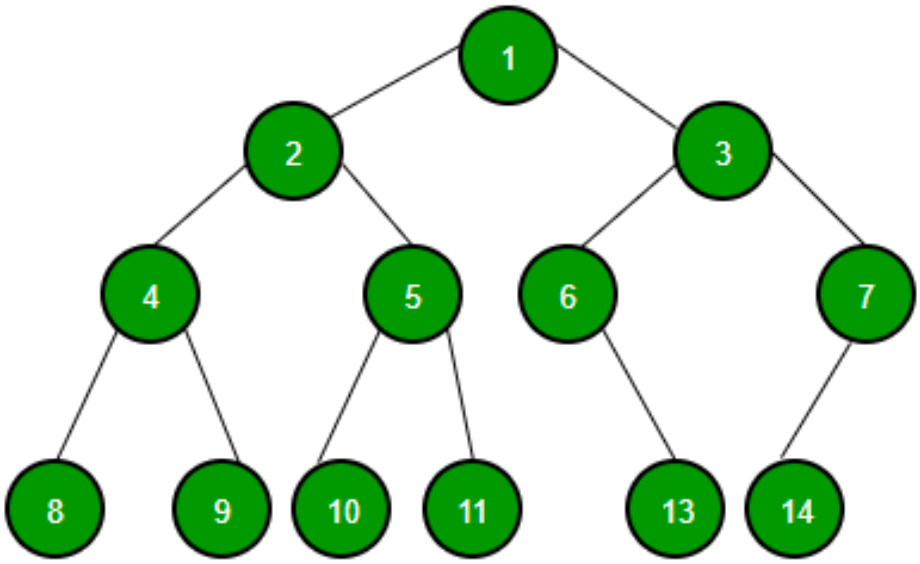
Their main advantage is the efficiency with which insert the insert operation can be carried out.

Hash Tables are usually implemented with built-in data structures like python dictionaries and can be used for a variety of things, from the implementation of an actual dictionary to entries of a NoSQL database.

2.6. Trees

Trees are non-linear data structures that can be considered an extension of a linked list. In a Tree, each node points to some "children" nodes or a null value, creating a hierarchical data structure.

Trees have a lot of properties, understand them we are going to take into consideration the Binary Tree in the image below (a Binary Tree is a Tree in which each node has at most 2 children).



Binary Tree. Image source: GeeksforGeeks

Properties and Terminology of Trees:

- **Node**: an element of the Tree, contains data and pointers to its children;
- **Edge**: The "link" between 2 nodes, every Tree has a maximum of $N - 1$ edges, where N is the number of nodes;
- **Parent Node**: the predecessor of a node, or the node that points to another. In the example above, among the others, "1" is the parent of "2" and "3", and "2" is the parent of "4" and "5";
- **Child node**: the descendant of a node. In the example above, among the others, "11" is a child of "2" and "3" is a child of "1";
- **Root Node**: The first element of the Tree and the only node without a parent node, in the example above the node is "1";
- **Siblings Nodes**: nodes that are children of the same parent node. In the example above, "4" and "5" and "8" and "9" are examples of siblings;
- **Leaf**: a node without children, like "11" or "14" in the example above.
- **Internal Node**: a node with at least 1 child. "7", "4" and "1" are internal nodes in the example above;
- **Degree**: the number of children that a node has. In a binary tree, this number is never greater than 2. The "Degree of Tree" is the Degree of the node with the highest Degree;
- **Level**: the "distance" of a Node from the Root Node, starting at 1. In the example above, the level of "1" is 1, the level of "3" is 2, the level of "5" is 3 and the level of "9" is 4;
- **Height**: is the "distance" between the furthest descending leaf and a node, starting at 0 for the leaves. In the example above, the height of " 10" is 0, the height of "4" is 1, the height of "3" is 2 and the Height of "1" is "4". The Height of the Root Node is also the Height of the Tree.
- **Depth**: the number of edges between a Node and the Root Node. For example, it is 0 for "1" and 2 for "7" in the Tree above.

Analysis of common operations on trees:

- Access:

  Since a Tree is not an indexed data structure, in the worst case it is possible that we need to look through all the nodes until we find the one we are looking for. For this reason, the time complexity of this operation is $O(n)$;

- Search:

  Trees are not ordered data structures, or at least not all of them. For this reason, searching in a tree could also mean that we need to search through all the other nodes, either with a Depth-First-Search or a Breadth-First-Search approach (we will look at both of these algorithms in the algorithms section of this portfolio) with a time complexity that in both cases is of $O(n)$;

- Insertion:

  Inserting a Node in a tree may require changing the positions of the other nodes as well to keep the properties of the tree. For this reason, this operation also has a time complexity of $O(n)$;

- Deletion:

  Deletion, just like insertion, may require rearranging all the other Nodes and so this operation also happens in $O(n)$ time complexity.

The space complexity of trees is $O(n)$, since they need to store a number of pointers that grows linearly with the number of nodes.

Given that there are a lot of subcategories of trees, they can be used in a lot of different ways. We will now look at some specific types of trees and discuss the real-world implementations of each.

2.6.1. Binary Trees

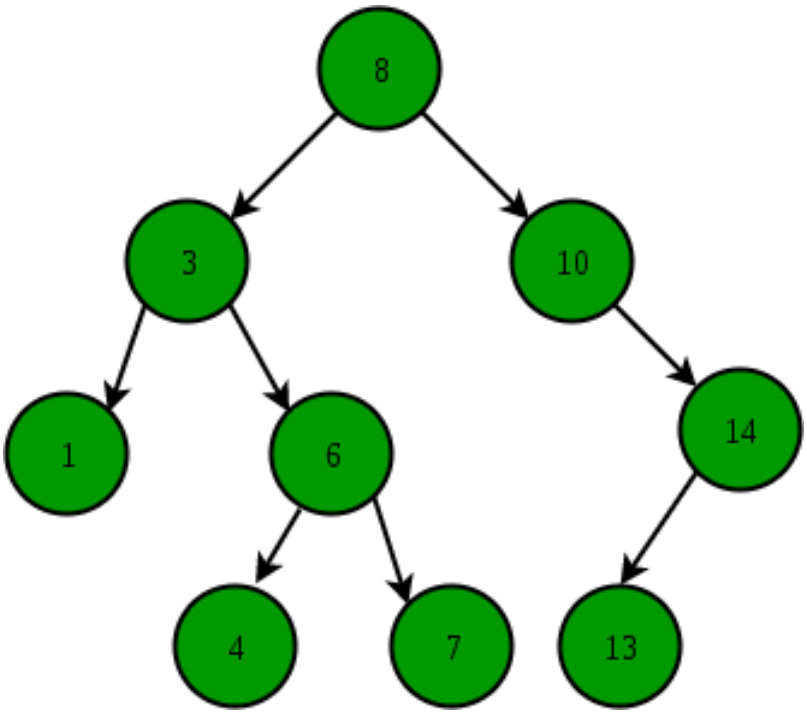
As I mentioned before, a Binary Tree is a Tree in which is node can have a maximum of 2 children, therefore each node contains some data, a pointer to its left child and a pointer to its right child.

A binary tree is not an ordered tree by definition, so the time complexity of common operations is the same as that of a normal tree.

There are, however, a lot of different implementations of Binary Trees that make them more efficient in those common operations:

2.6.1.1. Binary Search Trees

Binary Search Trees (BST) are Binary Trees in which the left child of a node (and all of its children) have a value smaller than that of the parent node and the right child of a node (and all of its children) have a value bigger than that of the parent node. As the name suggests, BSTs are used to implement the Binary Search algorithm on them.



Binary Tree. Image source: GeeksforGeeks

Analysis of common operations on BSTs:

- Search / Access:

  Binary Search Trees are not indexed data structure, so to access an element we need to search for it. BSTs are ordered data structures that work really well with the Binary Search algorithm (an analysis of this algorithm can be found in the algorithms part of this portfolio). Because of the fact that we can use Binary Search with this data structure, we can find an element with time complexity of $O(h)$, where h is also the height of the tree;

- Insertion:

  Inserting a Node in a BST may require us to change the order of all the other nodes, so in the worst-case scenario the time complexity of this operation will be $O(n)$, but in the average case the time complexity of this operation will depend on the height of the tree, so $\Theta(h)$;

- Deletion:

  Deleting a Node has the same impact as inserting a Node. the time complexity of this operation will therefore be $\Theta(h)$ in the average case and $O(n)$ in the worst.

An implementation of a BST could be using it together with a binary search algorithm.

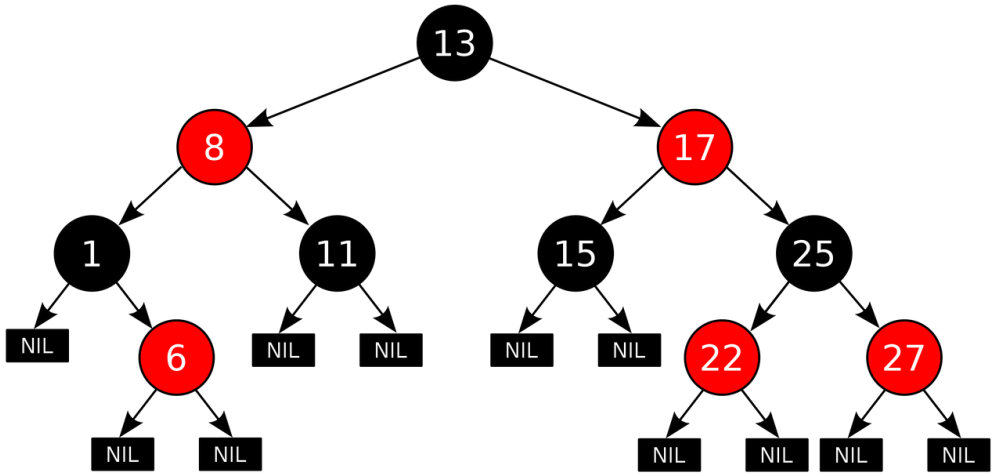
The main problem with BSTs is that they can be unbalanced when the nodes skew to one of the sides of the tree. In that case, the height of the tree is n and the operations in the tree become inefficient. To avoid this and have a height of $log(n)$ we can use a self-balancing Binary Tree.

2.6.1.2. Red-Black Trees

Red-Black trees are a common kind of self-balancing Binary Trees in which:

- Each node has a color property (either red or black);
- The root is always black;
- A Node cannot have the parent or children of its the same color, except if one or both of its children are leaves;
- Its leaves have a null value and are considered black;
- And every path from a node to any of its null descendants contains the same number of black nodes.

Red-Black Trees, just like BSTs, are used to implement the Binary Search algorithm on them, but these are usually more efficient.



Red Black Tree. Image source: Wikipedia

Analysis of common operations on Red-Black Trees:

- Search / Access:

  It is possible to perform a Binary Search on a Red-Black Tree, which as we have seen before allows us to search an element with a time complexity of $O(h)$. Since Red-Black trees are a balanced data structure, h will be $log(n)$ and therefore this operation will happen with time complexity of $O(log(n))$;

- Insertion:

  Since a Red-Black Tree is a balanced Tree, and the insertion operation depends on the height of the tree, we can say that this operation will have a worst-case time complexity of $O(log(n))$;

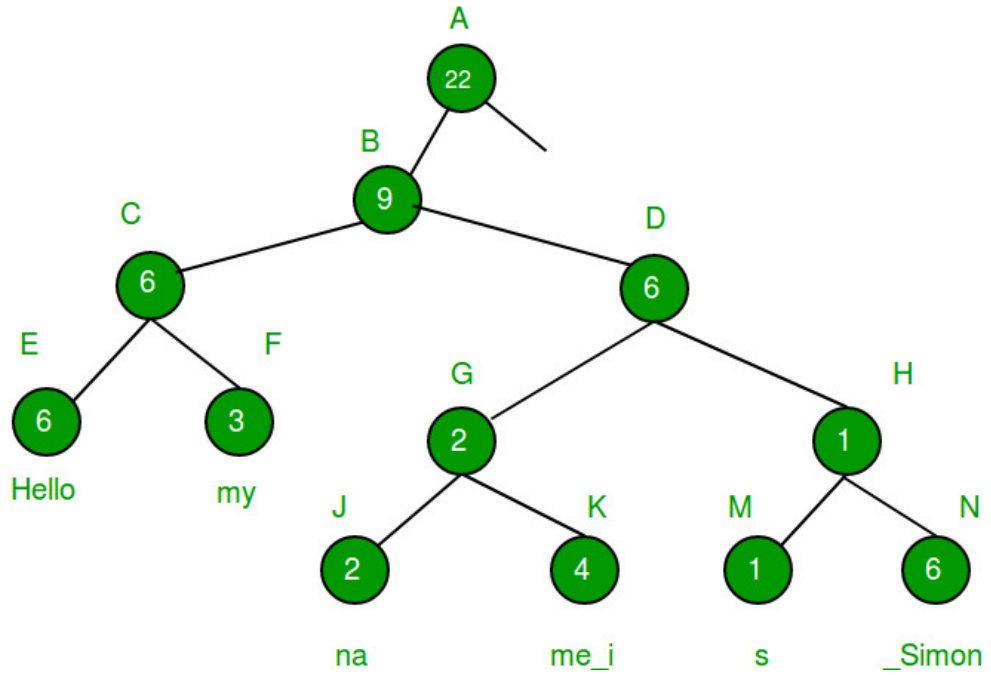
- Deletion:

  Deleting a node can have the same impact as inserting a node, and this operation depends on the height of the Tree too. Since the height of a Red-Black tree is $O(log(n))$, this operation will have a time complexity of $O(log(n))$.

2.6.1.3. Ropes

A Rope is a Binary Tree used for string manipulation. In a rope, each leaf holds a substring and each inner node the total length of the substrings that are descendants of its left child.

In the example below, the value of root node is the total length of the string, which is not a mandatory feature but can be useful in common operations, as we will see below.



Rope. Image source: GeeksForGeeks

Common operations that can be done on a rope are different than those done on other trees. These operations include:

- Index:

  Searching an element by their index is a very common operation in string manipulation, and thanks to the value of the inner nodes this operation can be done on Ropes with time complexity of $O(log(n))$.

  To understand how that is possible let's consider an example, finding the character with at the position $i = 8$ in the rope in the image above. We start by comparing i to the value of the root (which in this case holds the value of the total length), and we quickly determine if the index is part of the string. Since it is smaller than the value of A, we move to A's left child (B). we compare i to the value of B and since 8 is smaller than 9 we move to B's left child. We compare i to the value of C (6) and since 8 is bigger, we move to C's right child (F) and since we're moving to the right we update i to be $i - C$ (8-6 = 2), and since F is a leaf we access the child at position i (which is now 2) of the substring (y), which is the 8th character of the whole string.

- Concat:

  To concatenate 2 Ropes we just need to assign them to a new common root node with the value equal to the sum of the length of the substrings that descend from its new left child. This operation can be made in $O(1)$ time complexity, but computing the value for the new root node is an operation that has a time complexity of $O(h)$, where h is the height of the tree and is equal to $log(n)$ in a balanced tree.

- Split:

When splitting a string starting from an index we need to make a distinction between 2 major cases: we need to start splitting after the last character of a leaf, or we need to start splitting starting from a middle character of a leaf. If our case is the latter, we assign 2 children to the leaf (which becomes an inner node), the left one containing the character that we don't need to split, and the right one containing the characters that we need to split.

After finding the leaf from which we need to split the Rope, we separate the nodes at the right of that leaf and we fix the weight of the inner nodes that were ancestors of the nodes we separated. We then assign the split nodes to a new common root node.

At this point, it may be necessary to rebalance both Ropes.

This operation has a time complexity of $O(log(n))$ since it is the sum of the time complexities of the operations that are needed to complete this operation;

- Insert:

  Inserting a Rope in the middle of another Rope is an operation that can be done by splitting the original Rope, concatenate the Rope we need to insert, and then concatenate the right part of the node we originally split. Rebalancing the tree may be also required. The time complexity of this operation will be the sum of the time complexities of 1 split operation and 2 concatenation operations $(O(log(n))$;

- Delete:

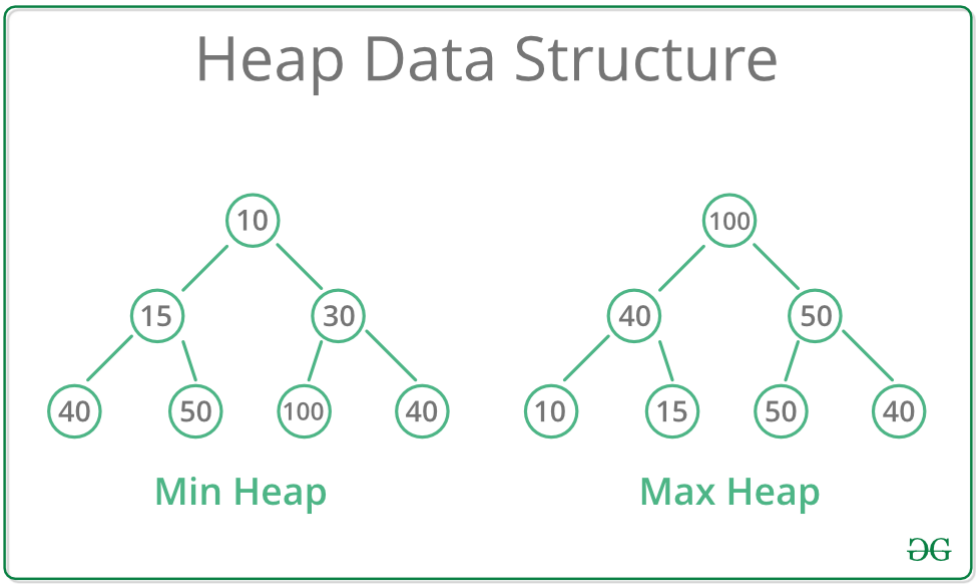
  To delete a substring at the middle of a Rope, we need to split the original rope starting at the first character that we want to delete. We then split the resulting right Rope starting after the last character that we need to delete, and we finally concatenate the left rope of the first split operation with the right Rope of the last split operation. This operation will also have a time complexity given by the sum of the operations that it uses, which will result in time complexity of $O(log(n))$.

Ropes are widely used in softwares that need to manage large texts, such as text editors and email clients, because of their performances in managing strings, especially compared with a traditional string implementation (an array of characters) and because they do not require contiguos memory allocation. Some of the disadvantages of ropes include the fact that they occupy more space than an array of string and their complexity, which can often lead to bugs.

2.6.2. Heaps

Heaps are trees in which the parent node always stores a value smaller than that of its children (in the case of a Min Heap) or bigger than that of its children (in case of a Max Heap). In this way, the root node always stores the smallest value (in a Min Heap) or the biggest value (in a Max Heap).

Heaps do not need to be Binary Trees, but they need to be complete (every level should have the maximum amount of nodes) and if they are not, new elements are added to the incomplete level from left to right. Because of this last property, Heaps are usually stored as arrays.



Heap. Image source: GeeksForGeeks

Analysis of common operations on Heaps:

- Search / Access:

  Searching an element that is not the root node (the node with the max value in a Max Heap or the node with a min value in a Min Heap), we may need to search through all the nodes to find the one we're looking for. Because of this, this operation will have a time complexity of $O(n)$;

- Insertion:

To insert an element in its correct position in a Heap we need to start by appending it to the last level, which can be done with average time complexity of $O(1)$ (if the heap is stored in an array). We then need to switch it with its parent node (in case the parent node is smaller and our heap is a Max Heap or in case the parent node is bigger and our Heap is a Min Heap) until it satisfies the properties of the heap. This second operation has a time complexity of $O(log(n))$;

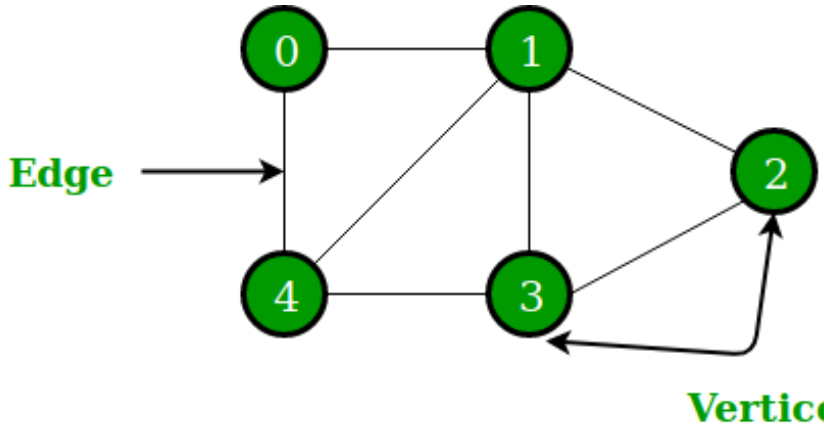
- Deletion:

To delete an element from a Heap we may also need to rearrange its nodes until the properties of the heap are satisfied. To do this we may need to switch an element for every level of the Heap and since the number of levels is given by $log(n)$ the time complexity of this operation will be $O(log(n))$.

Heaps are mainly used as an auxiliary data structure in various algorithms, like the Heapsort algorithm.

2.7. Graphs

Graphs are non-linear data structures made of vertices (or nodes) that store data and edges (that can also store data). Graphs are used to represent the relationships between its nodes or vertices. We have already examined a subset of Graphs, Trees.



Graph. Image source: GeeksForGeeks

Graphs terminology:

- **Vertex** (or Node), an element of the graph that always contains some data;
- **Edge**, the relationship between 2 vertices, can also contain information;
- **Adjacency**, two nodes connected via an edge;
- **Path**, a sequence of edges between 2 vertices;
- **Eulerian Path**, a path that visits every edge once (but can visit vertices more than once) and ends up in a vertex which is not the starting one;
- **Eulerian Cycle**: a path that visits every edge once (but can visit vertices more than once) and ends up in the starting vertex;
- **Hamiltonian Path**, a path that visits every vertex only once (but can visit edges more than once) and ends up in a vertex which is not the starting one;
- **Hamiltonian Cycle**: a path that visits every vertex only once (but can visit edges more than once) and ends up in the starting vertex;
- **Parallel Edges**, two or more edges that connect the same vertices;
- **Loop**, an edge that connects a node to itself.

Types of Graphs:

- **Finite**. A finite Graph contains a finite number of edges and vertices;
- **Infinite**. An infinite Graph contains an infinite number of vertices and edges;
- **Trivial**. A trivial Graph contains only one vertex and no edges;
- **Simple**. A simple Graph contains only one edge between a pair of vertices;
- **Non Simple**. A non-simple Graph contains more than one edge between a pair of vertices;
- **Multi-Graph**. A multi-graph contains some parallel edges but no loops;
- **Pseudo-Graph**. a pseudo-graph is a graph with at least a loop and a parallel edge;
- **Null**. A null graph contains vertices but no edges;
- **Complete** (or Full Graph). In a complete graph every vertex is adjacent to all the others;
- **Unweighted**. In an unweighted graph the edges do not store data;
- **Weighted**. In a weighted graph the edges store data;
- **Directed**. In a directed graph the edges connect 2 vertices only in one direction;
- **Undirected**. In an undirected graph the edges connect 2 vertices in both directions;
- **Topological**. In Topological Graphs, the vertices are represented by distinct points in space.

Ways of representing a graph:

- **Adjacency List:**

  With an Adjacency List, we use an array to store information about the graph. In an Adjacency List, the array element with the same index as the id of a vertex contains information about its adjacent vertices. An Adjacency List for the graph in the image above will look like this: `[[1,4],[0,2,3,4],[1,3],[1,2,4],[0,1,3]]` .

  Analysis of common operations on Adjacency Lists:

  - Storage:

    Storing a graph as an Adjacency List can be done with time complexity of $O(|V| + |E|)$, where V is the number of vertices and also the length of the array, and E is the number of edges;

  - Add Vertex:

    Adding a vertex to a graph represented as an Adjacency List can be done with time complexity of $O(1)$ since we just need to store a new element to the list;

  - Add Edge:

    Adding an edge to a graph represented as an Adjacency List can be done with time complexity of $O(1)$, since we just need to add to the arrays representing the two nodes 1 value;

  - Remove Vertex:

    Removing a vertex from a graph represented as an Adjacency List can be done with time complexity of $O(|V| + |E|)$, since we need to remove the edges to that vertex from all the other vertices as well;

  - Remove Edge:

    Removing an edge from a graph represented as an Adjacency List can be done with time complexity of $O(|E|)$, since we need to search and remove the edge from the list of edges of the two nodes that it connects.
    The space complexity of an Adjacency List is $O(|V| + |E|)$.

- **Adjacency Matrix:**

  With an Adjacency Matrix, we use a 2D matrix to store information about the graph. In an Adjacency Matrix, the array element with the same index as the id of a vertex contains an array that indicates if a vertex is connected to another or not (1 if it is, 0 if it is not). An Adjacency Matrix for the graph in the image above will look like this:

  ```
  [
  [0,1,0,0,1],
  [1,0,1,1,1],
  [0,1,0,1,0],
  [0,1,1,0,1],
  [1,1,0,1,0]
  ]
  ```

  Analysis of common operations on Adjacency Matrices:

- Storage:

  Storing a graph as an Adjacency matrix can be done with time complexity of $O(|V|^2)$, where V is the number of vertices, the number of arrays in the matrix, and the length of each array;

- Add Vertex:

  Adding a vertex to a graph represented as an Adjacency Matrix can be done with time complexity of $O(|V|^2)$ since we need to update all the arrays of which the matrix is made up as well as adding a new array;

- Add Edge:

  Adding an edge to a graph represented as an Adjacency Matrix can be done with time complexity of $O(1)$, since we just need to add to the update 2 values in the matrix;

- Remove Vertex:

  Removing a vertex from a graph represented as an Adjacency Matrix can be done with time complexity of $O(|V|^2)$, because we need to update all the other arrays of which the matrix is made up;

- Remove Edge:

  Removing an edge from a graph represented as an Adjacency Matrix can be done with time complexity of $O(1)$, since we need to just update 2 values in the matrix. The space complexity of an Adjacency Matrix is $O(|V|^2)$.

Because of their space complexity, it makes sense to use Adjacency Matrices either for small Graphs or Graphs with a lot of edges.

Graphs can be used everywhere we need to store relationships of any kind between elements. Since Trees are a subset of Graphs, all the real-world implementations of Trees are also real-world implementations of graphs. Another possible real-world implementation of Graphs is for road maps.

---

3. Algorithms

We have seen what is an algorithm and how to analyze one in the first part of the portfolio. In this section, we will analyze in-depth some common algorithms.

3.1. Searching Algorithms

Searching Algorithms are algorithms designed to find an element in the data structure for which the algorithm has been designed. We can find 2 main categories of Searching Algorithms:

- **Sequential Searching Algorithms**, designed to search for an item in unsorted data structures, check every item in the data structure;
- **Interval Searching Algorithms**, designed to search for an item in sorted data structures, only check some items of the data structure. This allows them to be more efficient than Sequential Searching Algorithms.

These algorithms are correct if they can correctly find an item in the data structure for which they were designed.

3.1.1. Linear Search

The Linear Search Algorithm is a Sequential Searching Algorithm.

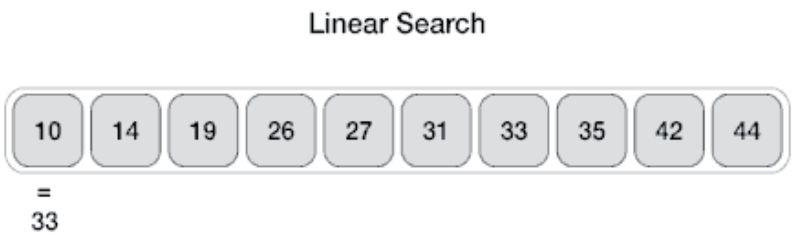
It takes in input an array and a value to search and iterates through the array to search for the value. It usually returns the index of the element (if it is in the array) or -1 (if the element is not in the array).

Here is a python example:

In [2]:
```
def linear_search(arr, val):
    for i in range(len(arr)):
        if (arr[i] == val):
            return i
    return -1

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
print(linear_search(array, 4))
```

```
7
```

This algorithm terminates for every input, either when it finds the element it is looking for, or when it iterates through the whole array. It can also be considered correct since it will always return the index of the element, or -1 if the element is not present.



Linear Search Animation. Gif source: tutorialspoint

Time Complexity Analysis

We have seen that this algorithm searches an element by iterating through the array, so in the case that the element is not present or is the last element of the array, it needs to go through the whole array of n elements. In the average case, the number of elements through which it needs to iterate also depend on n. Its time complexity in the average and worst-case will therefore be $\Theta = O(n)$.

In the best case, the element to search will be the first one, and thus the time complexity of this algorithm in the best case will be $\Omega(1)$.

Space Complexity Analysis

No auxiliary data structures are required by this algorithm, so the auxiliary space complexity will be $O(0)$ while the total space complexity will be $O(n)$.

3.1.2. Binary Search

The Binary Search Algorithm is an Interval Searching Algorithm.

It takes in input a sorted array and the value to search in it. It works by finding the median point of the array and returning the index of that point if that element contains the value we are searching for or recursively calls itself on the half of the array which could contain the element (the right part if the value of the main element was smaller than that of the value we are looking for or the left part if the value of the median element was greater). The process is recursively repeated until the element is found or there are no subarrays left to search.

Here is a python example:

In [3]:
```
def binary_search (arr, left, right, val):
    # check that the subarray is not empty
    if right >= left:
        # find the index of the median value
        mid = left + (right - left) // 2
        # return the median value if it is the one we were searching for
        if arr[mid] == val:
            return mid
        # if the median value is greater, recursively search the left part of
        elif arr[mid] > val:
            return binary_search(arr, left, mid-1, val)
        # otherwise recursively search the right part.
        else:
            return binary_search(arr, mid + 1, right, val)
    # if the subarray is empty return -1
    else:
        return -1
array = [0, 2, 5, 7, 8, 11, 12, 13, 19, 23, 26, 32, 41]
print(binary_search(array, 0, len(array), 26))
```

```
10
```

This algorithm terminates for every input, either when it finds the element and returns its index or when it tries to search an empty subarray and returns -1. Because of this, it can also be considered correct.

Binary Search Animation. Gif source: Brilliant

Time Complexity Analysis

Because at each recursive call we split the array of size n in half, it will take an amount of steps k to get to an array that will either contain 1 or 0 elements (depending on if the element we are searching for is in the array). Hence, we know that $n = 2^k$ and at this point we can easily calculate $k = log_2(n)$.

Because of this, the time complexity of this algorithm in its average and worst case is $\Theta = O(log(n))$.

In the best case, the element we are looking for is in the middle of the array and thus will be the first value we will search. So the time complexity of this algorithm in the best case will be $\Omega(1)$.

Space Complexity Analysis

No auxiliary data structures are required by this algorithm, so the auxiliary space complexity will be $O(0)$ while the total space complexity will be $O(n)$.

3.2. Sorting Algorithms

Sorting Algorithms are algorithms specifically designed to order elements in ascending or descending order, in the data structure for which the algorithm has been designed.

Sorting algorithms may vary a lot for both time and space efficiency, but also for their behavior in edge cases.

We can also distinguish between stable and unstable Sorting Algorithms: Stable Algorithms maintain the relative order of elements with the same value, while Unstable Algorithms do not. Every Unstable Algorithm can become Stable if we add the initial index of the element as the second sorting key.

We can say that a sorting algorithm is correct if it is capable of sorting the items in the data structure for which it has been designed.

3.2.1. Selection Sort

The Selection Sort Algorithm is an in-place, iterative sorting algorithm.

It works by comparing the first item in an array to all the other items with a greater index and swaps it with the smallest it finds. It then moves to the next element and repeats the process until it gets to the last item of the array. In this way, after each iteration, one more item is sorted.

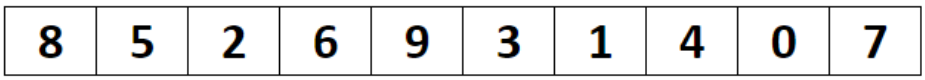
Here we can see a simple implementation in python:

In [4]:
```
def selection_sort(arr):
    for i in range(len(arr)-1):
        # find the index of the smallest element
        min_number_index = i
        for j in range(i, len(arr)):
            if arr[min_number_index] > arr[j]:
                min_number_index = j
        # swap the elements
        arr[i], arr[min_number_index] = arr[min_number_index], arr[i]

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
selection_sort(array)
print(array)
```

```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

Because this implementation of the algorithm swaps the elements in their correct positions, it does not maintain the relative order of the elements and it is therefore unstable.

We can see that this algorithm terminates for every input since its loops will stop at the end of the array. Even if this algorithm is not really time efficient, we know that its output will always be correct because it will compare each item in the array against all the others in the unsorted section of the array.



Selection Sort Animaiton. Blue = current index, Red = current minimum, Yellow = already sorted. Gif source: GitBooks

Time Complexity Analysis

As we can see from the code, the algorithm compares the element in the innermost of two nested loops, the outermost one will run n times, while the innermost loop will run $n-1$ times the first times it is called, and this number will decrease at each call until it runs just 1 time. The operations inside the innermost loop will run a number of times defined by the arithmetic series $\frac{n(n+1)}{2}$.

From the arithmetic series above we can easily find out that the dominant term is n^2, therefore, the average, best, and worst-case-scenario time complexity will be $\Theta = \Omega = O(n^2)$.

This means that this algorithm will take the same amount of steps also in every edge case (array already sorted, array sorted backward, array where all the items are the same).

Space Complexity Analysis

Because this is an in-place algorithm, it does not require any auxiliary space, so the auxiliary space complexity will be $O(0)$ while the total space complexity will be $O(n)$.

3.2.2. Bubble Sort

The Bubble Sort Algorithm is another example of an in-place, iterative sorting algorithm.

Here is how this algorithm works: it starts by comparing the first element in the array to the next. If the first element is bigger than the next, it swaps them before moving to the next element and repeats the process, until it gets to the end of the array. It then starts a new iteration with one less item to be compared, until the array is sorted.

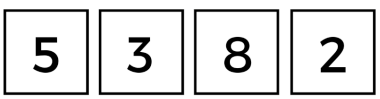
Here we can see a simple implementation in python:

In [5]:
```
def bubble_sort(arr):
    # check if any items have been swapped
    has_swapped = True
    # keep track of the index
    i = 0
    while(has_swapped and i<len(arr)):
        has_swapped = False
        for j in range(len(arr) - i - 1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                has_swapped = True
        i += 1

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
bubble_sort(array)
print(array)
```

```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

Because at each iteration this algorithm moves the item with the largest value at the end, the relative order of elements with the same value remains the same and thus is algorithm is stable.

This algorithm terminates for every input, as soon as there is an iteration in which no elements of the array need to be sorted.



Bubble Sort Animation. Gif source: Medium

Time Complexity Analysis

Just like the selection sort algorithm, this algorithm compares the element in the innermost of two nested loops, the outermost one will run n times, while the innermost loop will run $n-1$ times the first times it is called, and this number will decrease at each call until it runs just 1 time. The operations inside the innermost loop will run a number of times defined by the arithmetic series $\frac{n(n+1)}{2}$.

From the arithmetic series we will find out that the asymptotic growth of this algorithm is quadratic. The average and worst-case-scenario time complexity will be $\Theta = O(n^2)$.

Since the algorithm checks if any swaps have been performed at each iteration, in the case that the input array is already sorted or all the items in the array are the same (no item is greater than the previous one), the time complexity of this algorithm will be $\Omega(n)$.

If instead the input array is sorted backward, the time complexity will be $O(n^2)$, since it will need to perform swaps at each operation.

Space Complexity Analysis

Because this is an in-place algorithm, it does not require any auxiliary space, so the auxiliary space complexity will be $O(0)$ while the total space complexity will be $O(n)$.

3.2.3. QuickSort

The QuickSort Algorithm is a recursive sorting algorithm.

The algorithm starts by choosing a "pivot" (there are different ways to do this) and puts all the elements smaller than the pivot at its left and the elements bigger than the pivot at its right. In this way, the pivot is in the correct position and the operation is recursively repeated on the 2 sub-arrays, with new pivots. The operation continues until the subarrays contain only one element.

Here we can see an example of implementation in python:

In [6]:

```
def quicksort(arr):
    # auxiliary arrays
    less = []
    equal = []
    greater = []
    if len(arr) > 1:
        # pick the first element as pivot
        pivot = arr[0]
        for x in arr:
            if x < pivot:
                less.append(x)
            elif x == pivot:
                equal.append(x)
            elif x > pivot:
                greater.append(x)
        # recursively repeat the operation
        return quicksort(less)+equal+quicksort(greater)
    else:
        return arr

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
print(quicksort(array))
```

```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

In this implementation, the pivot is always the first item of the array/subarray. Other ways of choosing the pivot include:

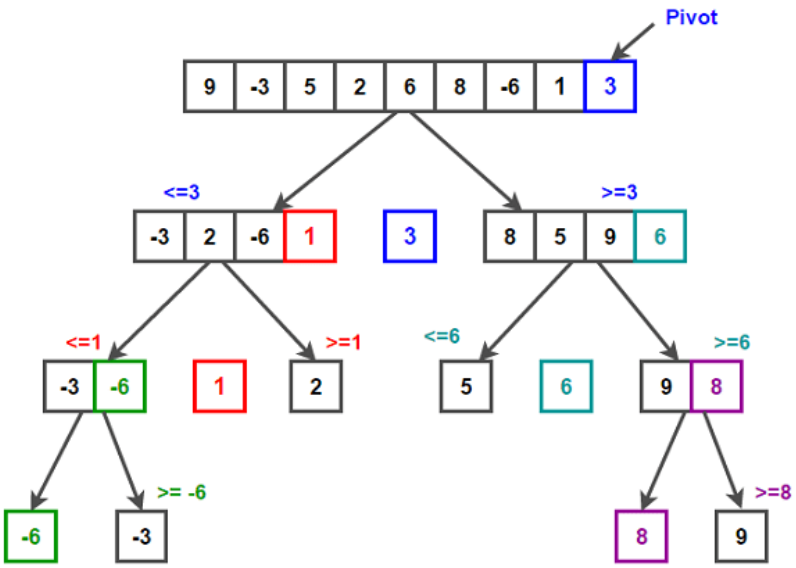
- picking always the last item in the array;
- picking a random value;
- picking three values (using one of the methods above) and choosing the median value as the pivot.

Because in this implementation we store the elements with the same value in a third array, we pick the pivot to be the first element of the subarrays, the relative order of the elements with the same value will not change. This specific implementation of QuickSort is stable, but it is not always the case.

This algorithm terminates for every input as soon as the subarrays contain 1 or 0 elements.

Time Complexity Analysis

We know that this algorithm will need to go through all the elements in the array, split it in half, and repeat the process on the halved arrays until it is left with subarrays of only one element. Assuming that the subarrays are of equal size, we can recursively split them in half $log_2(n)$ times. This happens because we know that it will take k steps to divide the array in subarrays of 1 element if at each step we halve the size of the array. So we can write the size of the array as $n = 2^k$ and at this point we can easily calculate $k = log_2(n)$.



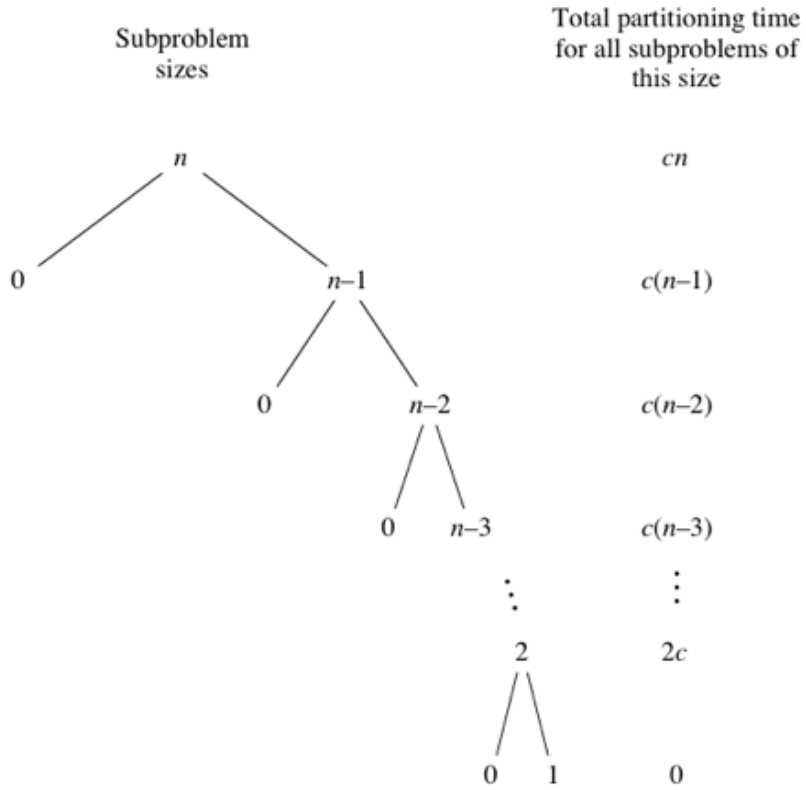
Quick Sort. Image Source: DeepAI

We can see in the image above how an array of 9 elements is recursively divided 3 times ($\approx log_2(9)$) before it is sorted. The number of comparisons in every iteration decreases as the size of the subarrays decreases, but we can see that it depends on the size of the array (n). Therefore, we will have in the average case a total number of operations, and thus its time complexity, that will grow in a quasilinear way ($\Theta(n \cdot log(n))$).

In case that the input array contains only elements with the same value, our implementation will only need to go through the array once, and since no element is bigger or smaller it will return two recursive functions called on empty arrays and the array of elements with the

same value as the pivot. Its best-case time complexity will therefore be $\Omega(n)$, but depending on how the algorithm is implemented, this may not be true in every case.

If instead the input array is already sorted or sorted backward, the time complexity of the algorithm will be $O(n^2)$, at least if the pivot is chosen to be always the first or always the last value of the array since we will have a situation similar to that of the image below, where after each "iteration" i, the array is not split enough but in an array $n - i - 1$ elements:



Quick Sort in worst-case time complexity. Image Source: Khan Academy

This situation can be avoided by picking a random pivot (which does not guarantee that this situation will not happen in other cases, although extremely unlikely especially for large enough inputs) or picking a median input.

Space Complexity Analysis

This implementation of the QuickSort algorithm requires an auxiliary space equal to the size of the array that it is sorting, so it will have an auxiliary space complexity of $O(n)$.

3.2.4. MergeSort

The MergeSort Algorithm is another example of a recursive sorting algorithm.

This algorithm works in a way that is conceptually really simple: it recursively divides an array in half until only arrays containing 1 element are left. It then starts to merge these arrays together while sorting them.

Here we can see a python implementation of this algorithm:

In [7]:
```
def merge(left, right):
    '''
```

```
    helper function to merge the array in an ordered way
    '''
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result

def mergesort(arr):
    # if the array contains only 1 element, return it
    if len(arr) < 2:
        return arr
    # otherwise recursively call this function to
    # the array in half...
    mid = len(arr) // 2
    left_arr = mergesort(arr[:mid])
    right_arr = mergesort(arr[mid:])
    # ...and then merge it
    return merge(left_arr, right_arr)

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
print(mergesort(array))
```

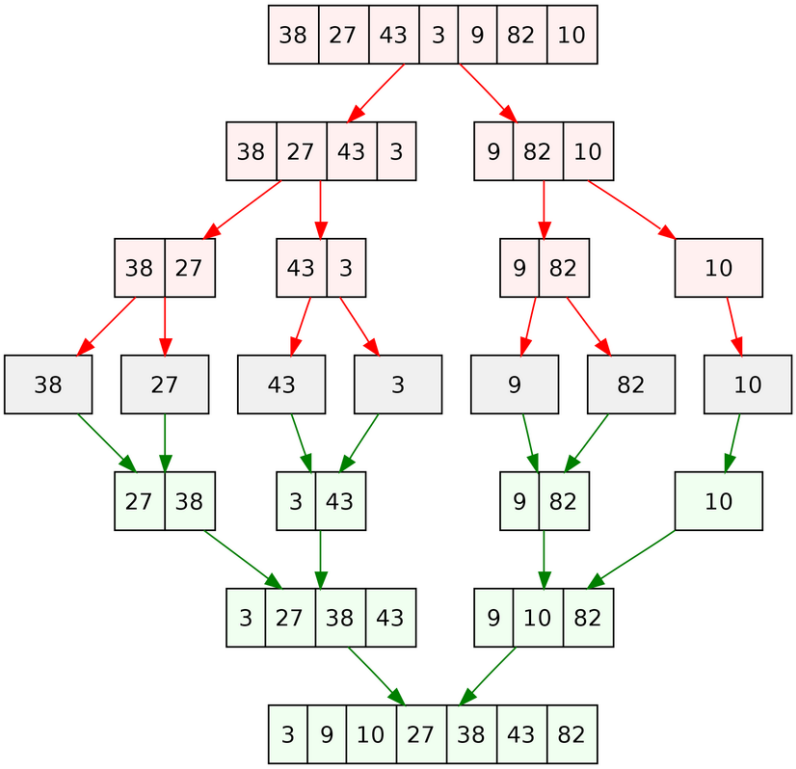
```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

While merging back the subarrays, this algorithm maintains the relative order of elements with the same value, and therefore it is a stable algorithm.

This algorithm will terminate for every input when the input is split into subarrays and then merged back together.

Time Complexity Analysis

This algorithm first needs to recursively divide the array into subarrays containing only 1 element, and then merge and order them.



Merge Sort. Image Source: Wikipedia

Splitting an array or subarray in half takse constant time, since in this case it is only a matter of computing the middle point of the array or subarray. As we have seen with the binary search, it takses k steps to divide the array into subarrays of 1 element each, so we know that $n = 2^k$ and we can easily calculate $k = log_2(n)$. The same amount of steps is taken to build the array back. In this case, however, at each step we will need to perform a number of comparisons that depends on the size of the array (n) to make sure that the array is sorted Because of this, the time complexity of this algorithm will be $\Theta(log(n))$.

An input array that is already sorted, sorted backward, or in which all items are the same will not affect the time complexity of this algorithm, therefore its best and worst-case time complexity will be $\Omega = O(n \cdot log(n))$

Space Complexity Analysis

The MergeSort Algorithm is not really space-efficient and requires an auxiliary space equal to the size of the array that it is sorting, so it will have an auxiliary space complexity of $O(n)$.

3.2.5. HeapSort

The HeapSort Algorithm is an iterative sorting algorithm.

Taken in input an unordered array, this algorithm turns that array into a Max Heap (n heap in which the value of a parent node is always bigger than that of its children and that can be represented as an array). It then swaps the last element in the heap with the first one (the root). Since the root is always the biggest element in a Max Heap, we know that that element is in its final position, and thus we can consider it to be in the sorted partition of the array. Then the algorithm "heapifies" (rearranges the elements so that the condition of the heap is met) the heap (or non-sorted partition of the array) and repeats the process with the last

element of the heap (or non-sorted partition of the array) until there are no more elements in the heap and the array is sorted.

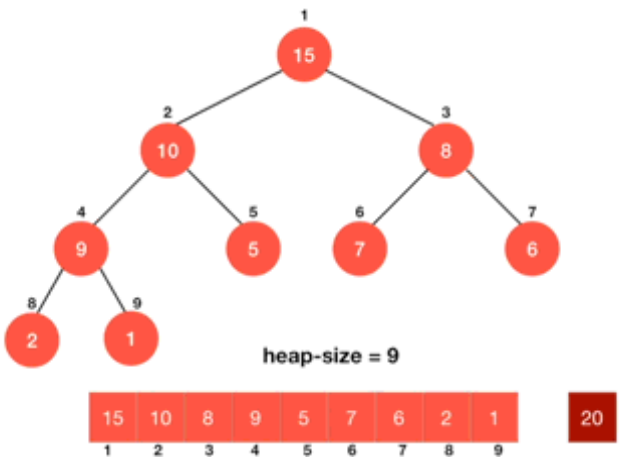
Here we can see a python implementation of this algorithm:

In [8]:
```
def heapify(heap, n, i):
    '''order the heap'''
    max = i
    left = 2 * i + 1
    right = 2 * i + 2
    # check if the left and right children exist.
    # If one of them is bigger than the parent, set it as max value.
    if left < n and heap[max] < heap[left]:
        max = left
    if right < n and heap[max] < heap[right]:
        max = right
    # swap the max value with the parent
    if max != i:
        heap[i], heap[max] = heap[max], heap[i]
        # heapify with the new parent node
        heapify(heap, n, max)

def build_max_heap(arr):
    """create an heap from an array"""
    for i in range(len(arr)//2 - 1, -1, -1):
        heapify(arr, len(arr), i)
    return arr

def heapsort(arr):
    build_max_heap(arr)

    # perform the heapsort
    for i in range(len(arr)-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

array = [9,6,3,5,3,0,2,4,3,7,8,2,6,1]
heapsort(array)
print(array)
```

```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

The initial order of the array is changed once the Max Heap is created, so the relative order of elements with the same value will change too. This algorithm is therefore unstable.

This algorithm will terminate for every input since it ends once its main loops gets to the first item of the heap.



Heap Sort Animation. Gif source: CodesDope

Time Complexity Analysis

We can start by analyzing the heapify function. In the worst-case scenario, this function will need to rearrange one element in each level of the tree. In the case of a binary heap, given that the heap is a balanced tree, its height will always be $h = log(n)$. Therefore, the worst-case time complexity of this function is $O(log(n))$.

The first step of this algorithm is turning the array into a Max Heap. Intuitively we may say that this operation will have a time complexity of $O(n \cdot log(n))$, which is correct but is not an asymptotic tight bound. Since the runtime of the heapify function is different for each node, depending on its level, the worst-case time complexity of the make_max_heap function will be $O(n)$. A complete analysis of this operation can be found here.

We can see in the code implementation above that once turned the array into a heap, we need to call the heapify function $n - 1$ times. For this reason, the time complexity of this algorithm in the best, average, and worst-case will be $\Omega = \Theta = O(n \cdot log(n))$.

If we pass as input an array that is already sorted, sorted backward, or in which all the items have the same value, the time complexity of this algorithm will not change.

Space Complexity Analysis

No auxiliary data structures are required by this algorithm, so the auxiliary space complexity will be $O(0)$ while the total space complexity will be $O(n)$.

3.2.6. Counting Sort

Counting Sort is a non-comparison, iterative sorting algorithm.

Unlike all the other sorting algorithms we have seen, the Counting Sort algorithm works only on positive integers in a range k.

Here is how the algorithm works:

1. It counts the number of occurrences of each unique value v in the array and stores the number of occurrences at index v in an auxiliary array;
2. It then performs a cumulative count of the elements in the auxiliary array (e.g.: `[0,2,1,1]` becomes `[0,2,3,4]` after a cumulative count) so that each value c in the auxiliary array is the last index + 1 at which the value v (given by the index of c) will appear in the output array;
3. Then, starting with the last element e in the initial array, it decreases the cumulative count of the element in position e in the auxiliary array, and inserts e at the correct index (the value stored in the position e of the auxiliary array) in the output array. In the end, it copies the output array in the original array.

Here is a python implementation of this algorithm where the range k is fixed to be 0-9:

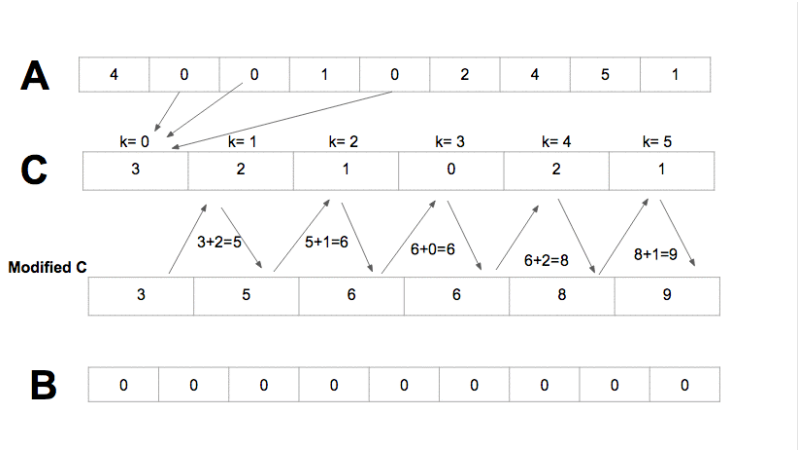
In [9]:
```
def counting_sort(array):
    size = len(array)
    output = [0] * size
    # Initialize count array
```

```
    aux = [0] * 10
    # Store the count of each element in count array
    for i in range(0, size):
        aux[array[i]] += 1
    # Store the cumulative count
    for i in range(1, 10):
        aux[i] += aux[i - 1]
    # Decrease the cumulative count by one for each position
    # Find the index of the element of the original array in aux array
    # Place the elements in output array, at the right index
    for i in range(size - 1, -1, -1):
        aux[array[i]] -= 1
        index = aux[array[i]]
        output[index] = array[i]

    # Copy the sorted element into original array
    for i in range(0, size):
        array[i] = output[i]

array = [9, 6,3,5,3,0,2,4,3,7,8,2,6,1]
counting_sort(array)
print(array)
```

```
[0, 1, 2, 2, 3, 3, 3, 4, 5, 6, 6, 7, 8, 9]
```

This algorithm is also stable because it maintains the relative order of elements with the same value when the array is sorted from the initial to the output array.

This algorithm will also terminate for every input as soon as the loops are executed.



Counting Sort Animation. Gif source: Brilliant

Time Complexity Analysis

We can analyze each step of this algorithm to find its time complexity:

1. Counting the number of occurences can be done with time complexity of $O(n)$, where n is the length of the array;
2. Performing a cumulative count can be done with time complexity of $O(k)$, where k is the size of the range of the positive integer values of which the array is made of, and the size of the auxiliary array;
3. Arranging the elements in ascending order in the output array can be done with time complexity of $O(n)$.

Since we have some operations that require a time complexity of $O(n)$ and another that requires a time complexity of $O(k)$, we can say that the time complexity of this algorithm is

$O(n+k)$.

In case that all the elements in the input array have the same value, the size of the range of positive integers k in the array will be 1, and thus its best-case time complexity will be $\Omega(n)$. If instead the input array is already sorted or sorted backward, it won't affect the time complexity of this algorithms, so its average and worst-case time complexity will be $\Theta = O(n+k)$.

Space Complexity Analysis

The auxiliary space required by this algorithm depends on the size of the array (n) and the size of the range of positive integers that the array contains (k). Its space complexity will therefore be $O(n+k)$.

3.3. Graph Algorithms

Some algorithms are designed to perform some specific operations on graphs or on given subcategories of them. Some common operations carried out by graph algorithms include traversal (visiting all the vertices in a graph) and pathfinding (finding the shortest path between 2 vertices).

3.3.1. Depth-First Search

The Depth-First Search Algorithm (DFS) is a graph traversal algorithm. It can be implemented on both trees and graphs, and it can also be modified and used for searching or pathfinding on these data structures. We will see how the algorithm works when used for graph traversal.

Starting from a given vertex (or the root node in the case of a tree), this algorithm visits all the possible vertices in a specific branch before backtracking.

The algorithm works by adding the vertex that is passed as an argument, v, to a data structure (usually a stack) – to keep track of the vertices that have already been visited. It then recursively calls itself on every neighbor of v that has not been visited yet, so that all the branches are visited. Finally, it returns the list of visited vertices.

Here is a python implementation of DFS used for graph traversal on a graph represented by an adjacency list in which each key of the dictionary is a vertex and its value is an array containing all the nodes to which it is connected:

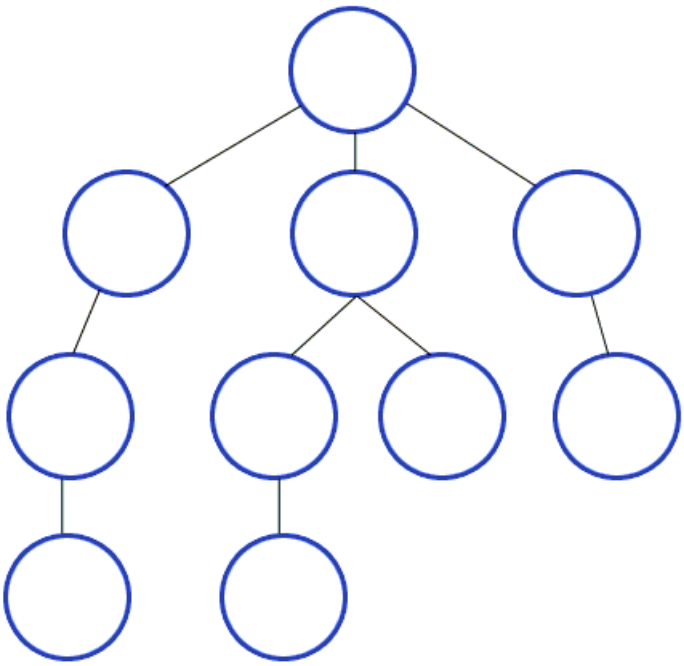
In [10]:
```
def dfs(graph, start, visited=None):
    if visited == None:
        visited = []
    visited.append(start)
    for vertex in graph[start]:
        if vertex not in visited:
            dfs(graph, vertex, visited)
    return visited

graph = {'A': ['B', 'D'],
         'B': ['A', 'C', 'D'],
         'C': ['B', 'F'],
         'D': ['A', 'B', 'E', 'F'],
```

```
        'E': ['D'],
        'F': ['C', 'D']}

print(dfs(graph, 'C'))
```

```
['C', 'B', 'A', 'D', 'E', 'F']
```

This algorithm will not terminate if implemented for graph traversal on an infinite graph (and could not terminate even if implemented for searching on an infinite graph). Otherwise, it will terminate once it has visited each vertex (or found a given vertex/path, depending on its implementation).

We know that this algorithm is correct if it can correctly traverse a graph.



Depth-First Search on a tree. Gif source: Wikimedia

Time Complexity Analysis

We have seen that this algorithm needs to visit all the vertices, starting at the initial one, checking all of its neighbors (the vertices connected to it by an edge), and recursively call itself on the neighbors that have not yet been visited. Therefore, the total number of operations depends on the number of vertices V and the number of edges E and its time complexity, in the average and worst case, will be $\Theta = O(|V| + |E|)$.

When this algorithm is implemented on a tree (which is a graph with $v - 1$ edges), its time complexity will be $\Omega(V)$.

Space Complexity Analysis

The auxiliary space required by this algorithm depends on the number of vertices, so its space complexity will be $O(V)$

3.3.2. Breadth-First Search

The Breadth-First Search Algorithm (BFS) is another graph traversal algorithm. Just like DFS, it can be implemented on both trees and graphs, and it can be modified and used for searching or pathfinding on these data structures. As for the DFS, we will see how the algorithm works when used for graph traversal.

Starting from a given vertex (or the root node in the case of a tree), this algorithm visits all of its neighbors stores them in a queue, and then repeats the process with the first vertex in the queue.

The algorithm works by storing the start node in a queue (to keep track of the vertices whose neighbors haven't been visited) and in another data structure (to keep track of the vertices that have been visited). Then, while there are items in the queue, it pops the first item from the queue and checks its neighbors, and those who haven't been visited yet are added to both the queue and the other data structure. In the end, it returns the data structure containing the visited nodes.

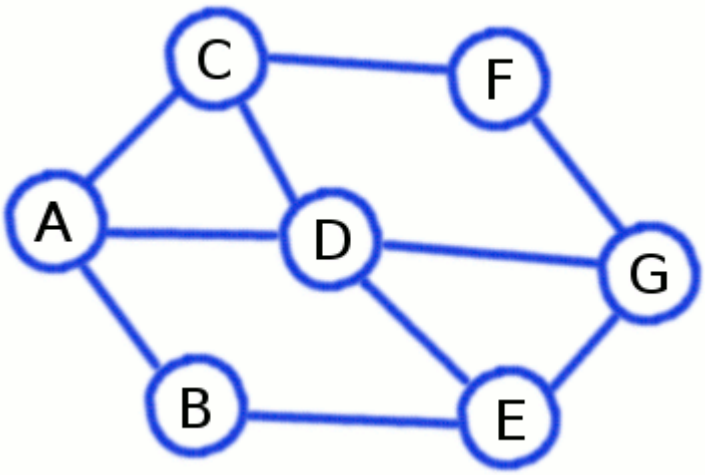
Here is a python implementation of BFS used for graph traversal on a graph represented by an adjacency list in which each key of the dictionary is a vertex and its value is an array containing all the nodes to which it is connected:

In [11]:
```
def bfs(graph, start):
    visited = []
    queue = []
    visited.append(start)
    queue.append(start)
    while queue:
        current = queue.pop(0)
        for vertex in graph[current]:
            if vertex not in visited:
                visited.append(vertex)
                queue.append(vertex)
    return visited

graph = {'A': ['B', 'C', 'D'],
         'B': ['A', 'E'],
         'C': ['A', 'D', 'F'],
         'D': ['A', 'C', 'E', 'G'],
         'E': ['B', 'D', 'G'],
         'F': ['C', 'G'],
         'G': ['D', 'E', 'F']}

print(bfs(graph, 'A'))
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

This algorithm will not terminate if implemented for graph traversal on an infinite graph (but unlike DFS, it terminates if implemented for searching on an infinite graph). Otherwise, it will terminate once it has visited each vertex (or found a given vertex/path, depending on its implementation).

We know that this algorithm is correct if it can correctly traverse a graph.



Breadth-First Search on a graph. Gif source: CodeAbbey

Time Complexity Analysis

We have seen that this algorithm needs to visit all the vertices, starting at the initial one, checking all of its neighbors (the vertices connected to it by an edge), and repeat the operation on the neighbors that have not yet been visited. Therefore, like for the DFS, the total number of operations depends on the number of vertices V and the number of edges E, and its time complexity, in the average and worst case, will be $\Theta = O(|V| + |E|)$.

When this algorithm is implemented on a tree (which is a graph with $v - 1$ edges), its time complexity will be $\Omega(V)$.

Space Complexity Analysis

The auxiliary space required by this algorithm, like in the case of the DFS, depends on the number of vertices, so its space complexity will be $O(V)$.

3.3.3. Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm used to find the shortest distance between two vertices in a weighted graph. This algorithm requires the weight of the edges to be positive to work. It can also be modified to find the distance between one vertex and all the others.

These are the steps of this algorithm:

1. Initialize 3 auxiliary data structures to keep track of the vertices to visit (starting from the starting vertex), from which adjacent vertex a vertex has been visited (None for the starting vertex); and the cost to get from the starting vertex to that one (0 for the starting vertex). We will call these data structures a, b, and c respectively for simplicity;
2. Set the current vertex to be the one with the lowest "cost" in a and remove it from a (in the first iteration it will be the starting vertex);
3. If the current vertex is the vertex to find, reconstruct and return the path to that vertex;
4. Otherwise, for each of its neighbors n, calculate the "cost" to get there and if they are not in c, or if they are but this new "cost" is lower than the previous one: update its value in c, add it to a with its new "cost" as priority, and set it as visited from the current node in b.
5. Repeat steps 2, 3, and 4 until there are elements in a or as long as you don't have found the vertex to find.

This algorithm is optimal, which means that it will always find the best solution.

Here is a python implementation of Dijkstra's Algorithm on a graph where each vertex has a set of spatial coordinates and a list of neighbors and the heuristic function returns the distance between 2 vertices. To keep track of the vertex to visit we will use a Priority Queue implemented with a Min-Heap.

In [12]:

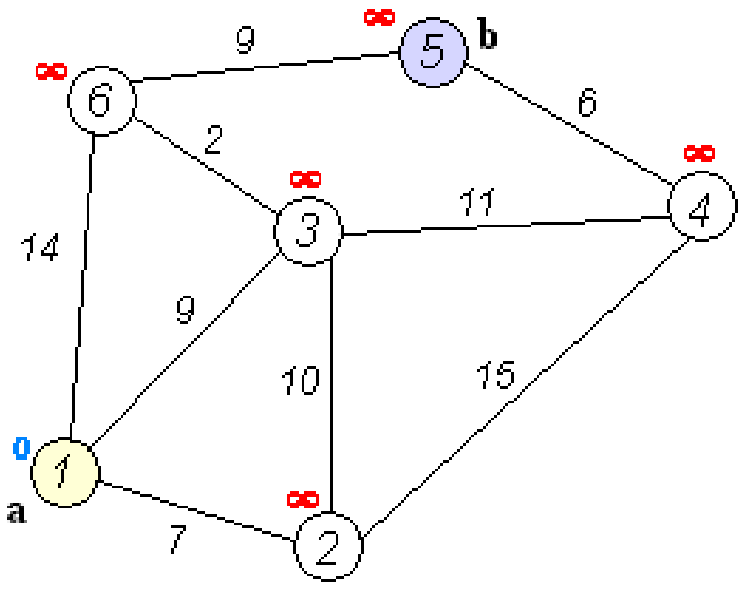
```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return not self.elements

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

class Graph():
    def __init__(self, graph):
        '''
        Initialize a graph, represented as an adjacency list
        '''
        self.graph = graph

    def dijkstra(self, start, to_find):
        to_visit = PriorityQueue()
        to_visit.put(start, 0)
        # Dictionary in which we store from which vertex another vertex has b
        from_v = {}
        # Dictionary in which we store the cost to get to a vertex
        cost_to_vertex= {}
        # Initialize the start node in both the from_v and cost_to_vertex dic
        from_v[start] = None  # It is none because it is the first node
        cost_to_vertex[start] = 0 # It is 0 because it is the first node

        while not to_visit.empty():
            # Get the vertex with the least cost in the queue
            current = to_visit.get()
            # If the vertex is the one to find, return the path to that verte
            if current == to_find:
                path = []
                while current != start:
                    path.append(current)
                    current = from_v[current]
                path.append(start)
                path.reverse()
                return path, "The cost is " + str(cost_to_vertex[to_find])

            for neighbor, weight in self.graph[current]["neighbors"]:
                # determine the new cost
                new_cost = cost_to_vertex[current] + weight
                # If the vartex hasn't been explored or is being explored fro
                if neighbor not in cost_to_vertex or new_cost < cost_to_verte
                    cost_to_vertex[neighbor] = new_cost
                    # calculate the priority
                    priority = new_cost
                    # add it in the queue
                    to_visit.put(neighbor, priority)
```

```
                    # add it to the path, assign it as value the node current
                    from_v[neighbor] = current

        return


graph = {
    "JFK" : {"coords": (40.730, -73.935), "neighbors": [("LAX", 92), ("LHR",
    "BER" : {"coords": (52.520, 13.404), "neighbors": [("FCO", 33), ("LHR", 5
    "FCO" : {"coords": (41.773, 12.239), "neighbors": [("BER", 33), ("LHR", 9
    "LHR" : {"coords": (51.509, 0.118), "neighbors": [("JFK", 427),("BER", 54
    "LAX" : {"coords": (34.052, -118.24), "neighbors": [("JFK", 92), ("HNL",
    "GRU" : {"coords" : (-23.588, -46.658), "neighbors" : [("JFK", 419), ("LH
    "JNB" : {"coords" : (-26.134, 28.240), "neighbors" : [("FCO", 351), ("GRU
    "SVO" : {"coords" : (55.751, 37.618), "neighbors" : [("BER", 122), ("PEK"
    "SYD" : {"coords" : (-33.947, 151.179), "neighbors" : [("LAX", 890), ("IC
    "PEK" : {"coords" : (40.072, 116.597), "neighbors" : [("SVO", 621), ("SYD
    "DEL" : {"coords" : (28.644, 77.216), "neighbors" : [("FCO", 358), ("PEK"
    "HNL" : {"coords" : (21.315, -157.858), "neighbors" : [("LAX", 129), ("SY
    "ICN" : {"coords" : (37.532, 127.024), "neighbors" : [("LAX", 567), ("SYD
}
g = Graph(graph)
start = "LAX"
end = "BER"
print(g.dijkstra(start, end))
```

```
(['LAX', 'JFK', 'LHR', 'BER'], 'The cost is 573')
```

This algorithm will terminate after v iterations, where v is the number of vertices in the graph.

The algorithm needs to correctly find the shortest path from one vertex to all the others to be correct.



Dijkstra's algorithm animation. Gif source: Wikipedia

Time Complexity Analysis

We can analyze each step of this implementation of the algorithm to find its overall time complexity:

1. Initializing the auxiliary data structures is done in constant time;

2. Getting the current node from the heap can be done with a worst-case time complexity of $O(log(V))$
3. Returning the value can happen in constant time, but generally depends on the size of the path, which can never be more than V;
4. We need to check the cost to all the neighbors of the current vertex to update the priority queue, so this operation will be dependant on the number of edges $O(E)$;
5. Repating the steps 2, 3, and 4 until we have found the correct vertex means that we will need to extract the current vertex from a priority queue that in the worst case contains all the vertices, an operation that can be done with a time complexity of $O(log(V))$. In the worst-case scenario, this operation will be repeated E times.

The overall worst-case time complexity of this specific implementation of Dijkstra's Algorithm will therefore be $O(|V|) + O(|E| \cdot log(|V|)) = O(|V| + |E| \cdot log(|V|))$.

Space Complexity Analysis

This implementation of the algorithm uses two auxiliary data structures whose length depends on the number of vertices. The space complexity of this algorithm will therefore be $O(V)$.

3.3.4. A* Algorithm

The A* Algorithm is an informed search algorithm. An informed search algorithm uses some additional information to determine its best next move.
This additional information is calculated in this case by a function $f(n) = g(n) + h(n)$, where n is the next vertex, $g(n)$ is the cost to get from the start vertex to the next, and $h(n)$ is an heuristic function that approximates the cost to get from n to the vertex to find, and never overestimates the real cost between n and the vertex to find. The algorithm will then pick its next move based on which one has the lowest value of $f(n)$.

Here is how this algorithm works:

1. Initialize some additional data structures to keep track of: the vertices to visit (starting from the starting vertex); from which adjacent vertex a vertex has been visited (None for the starting vertex); and the cost to get from the starting vertex to that one (0 for the starting vertex). We will call these data structures a, b, and c respectively for simplicity;
2. Set the current vertex to be the one with the lowest value of $f(n)$ in a and remove it from a;
3. If the current vertex is the vertex to find, reconstruct and return the path to that vertex;
4. Otherwise, for each of its neighbors n, calculate $g(n)$ and if they are not in c, or if they are but this new value of $g(n)$ is lower than the previous one: update its value in c, add it to a with priority $f(n)$, and set it as visited from the current node in b.
5. Repeat steps 2, 3, and 4 until there are elements in a or as long as you don't have found the vertex to find.

This algorithm is optimal, which means that it will always find the best solution.

Here is a python implementation of the A* algorithm, on a graph where each vertex has a set of spatial coordinates and a list of neighbors and the heuristic function returns the distance

between 2 vertices. To keep track of the vertex to visit we will use a Priority Queue implemented with a Min-Heap.

In [13]:

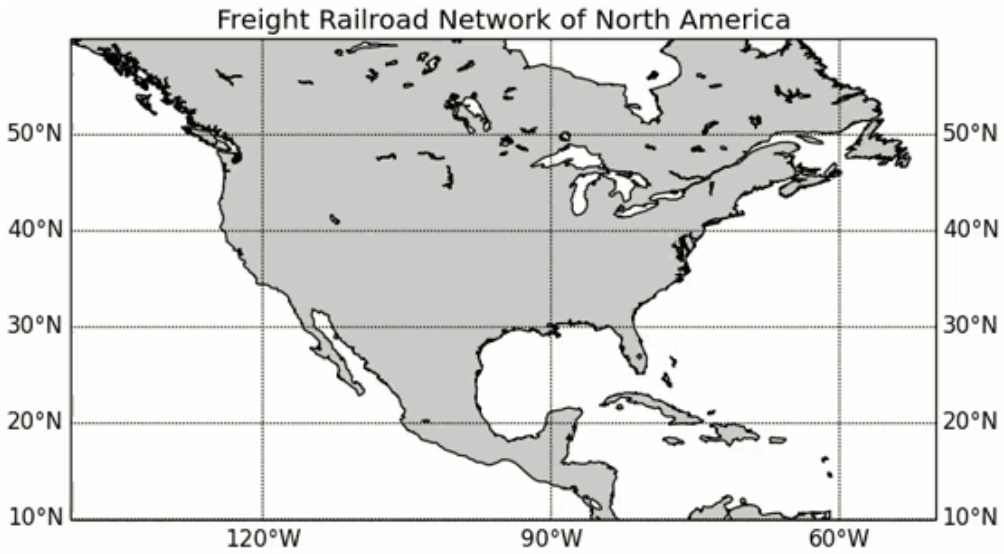
```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return not self.elements

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

class Graph():
    def __init__(self, graph):
        '''
        Initialize a graph, represented as an adjacency list
        '''
        self.graph = graph

    def heuristic(self, current, to_find):
        (x1, y1) = self.graph[current]["coords"]
        (x2, y2) = self.graph[to_find]["coords"]
        dist =  abs(x1 - x2)**2 + abs(y1 - y2)**2
        return dist

    def astar(self, start, to_find):
        to_visit = PriorityQueue()
        to_visit.put(start, 0)
        # Dictionary in which to each key corresponds the previous vertex
        from_v = {}
        # Dictionary in which to each key corresponds the cost to get there
        cost_to_vertex= {}

        # Initialize the start node in both the from_v and cost_to_vertex dic
        from_v[start] = None  # It is none because it is the first node
        cost_to_vertex[start] = 0 # It is 0 because it is the first node

        while not to_visit.empty():
            # Get the vertex with the most priority from the queue
            current = to_visit.get()
            # If the vertex is the one to find, return the path to that verte
            if current == to_find:
                path = []
                while current != start:
                    path.append(current)
                    current = from_v[current]
                path.append(start)
                path.reverse()
                return path, "The cost is " + str(cost_to_vertex[to_find])

            for neighbor, weight in self.graph[current]["neighbors"]:
                # determine the new cost
                new_cost = cost_to_vertex[current] + weight
                # If the vartex hasn't been explored or is being explored fro
                if neighbor not in cost_to_vertex or new_cost < cost_to_verte
                    # update the cost
                    cost_to_vertex[neighbor] = new_cost
                    # calculate the priority
```

```
                    priority = new_cost + self.heuristic(neighbor, to_find)
                    # add it in the queue
                    to_visit.put(neighbor, priority)
                    # add it to the path, assign it as value the node current
                    from_v[neighbor] = current

        return "No path"


graph = {
    "JFK" : {"coords": (40.730, -73.935), "neighbors": [("LAX", 92), ("LHR",
    "BER" : {"coords": (52.520, 13.404), "neighbors": [("FCO", 33), ("LHR", 5
    "FCO" : {"coords": (41.773, 12.239), "neighbors": [("BER", 33), ("LHR", 9
    "LHR" : {"coords": (51.509, 0.118), "neighbors": [("JFK", 427),("BER", 54
    "LAX" : {"coords": (34.052, -118.24), "neighbors": [("JFK", 92), ("HNL",
    "GRU" : {"coords" : (-23.588, -46.658), "neighbors" : [("JFK", 419), ("LH
    "JNB" : {"coords" : (-26.134, 28.240), "neighbors" : [("FCO", 351), ("GRU
    "SVO" : {"coords" : (55.751, 37.618), "neighbors" : [("BER", 122), ("PEK"
    "SYD" : {"coords" : (-33.947, 151.179), "neighbors" : [("LAX", 890), ("IC
    "PEK" : {"coords" : (40.072, 116.597), "neighbors" : [("SVO", 621), ("SYD
    "DEL" : {"coords" : (28.644, 77.216), "neighbors" : [("FCO", 358), ("PEK"
    "HNL" : {"coords" : (21.315, -157.858), "neighbors" : [("LAX", 129), ("SY
    "ICN" : {"coords" : (37.532, 127.024), "neighbors" : [("LAX", 567), ("SYD
}
g = Graph(graph)
print(g.astar("LAX", "BER"))
```

```
(['LAX', 'JFK', 'LHR', 'BER'], 'The cost is 573')
```

This algorithm will terminate for every input, either because it finds the shortest path to a vertex or because no elements are left to visit, and no path exists.

This algorithm is correct if it can find the shortest path between 2 vertices in a graph.



A* algorithm animation in a real-world scenario: finding the shortest path from Washington, DC to Los Angeles, CA. Gif source: Wikipedia

Time Complexity Analysis

In general, the time complexity of this algorithm depends on the number of vertices that need to be visited before finding the vertex that we were looking for and thus on the efficiency of the heuristic function.

With an uninformative heuristic function, this algorithm will behave like Dijkstra's algorithm, and potentially visit every vertex and edge in the graph, an operation that has a time complexity of $O(|V| + |E|)$. This time complexity can also be expressed as $O(b^d)$, where b is the average number of adjacent vertices and d is the number of edges between the starting node and the node to find.

In the best case for this algorithm, the heuristic function is so efficient that only the nodes in the path between the two vertices are ever explored, its time complexity, in this case, will be $\Omega(d)$, where d is the number of edges between the 2 nodes.

Space Complexity Analysis

This Algorithm keeps track of all the vertices it visits, so in the worst case, its space complexity will be of $O(V)$.

3.4. Graph Algorithms In Action

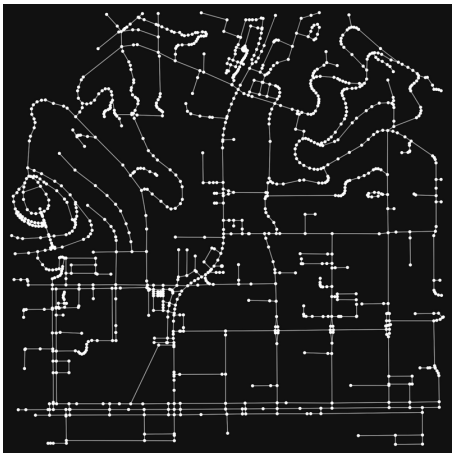
One common use of graph algorithms is finding the shortest route on a street network. In this section, we will look at one way to represent street networks with graphs, how to use the Dijkstra's and the A* algorithms to find the shortest path, and then compare their efficiency.

3.4.1. Street Networks In Code

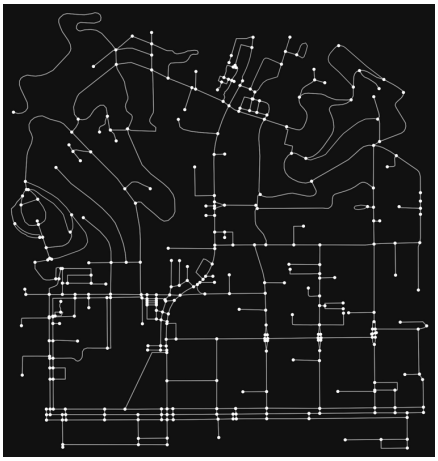
To understand how we can represent streets with code we need to introduce the concept of network.
While a graph is an abstract mathematical representation of elements and their connections, a network may be thought of as a real-world graph. Networks inherit the terminology of graph theory. [...]. A complex network is one with a nontrivial topology (the configuration and structure of its nodes and edges) – that is, the topology is neither fully regular nor fully random (Boeing, 2017).
According to this definition, a complex network seems like a good choice for our case, but we also want our graph to have nodes and edges ebedded in space. A complex network with nodes and edges embedded in space is called a complex spatial network, and that is what we will use.

We will look at different examples and implementations that are specific to python and Geoff Boeing's OSMnx library. There may be other ways to represent and interact with street networks but for the purposes of this portfolio this one should be enough. All the maps are downloaded by the library from OpenStreetMap.

A street network is represented by the OSMnx library, or in general by OpenStreetMap, with a series of vertices that represent intersections, the beginning or end of a road, or a change in the topography of a road.
The properties of each vertex include: a unique id, a set of coordinates, the type of road to which it belongs, optionally the max speed (in mph or km/h) and the road name, a list of adjacent vertices with all of their properties and the distance in meters from the current vertex. OSMnx uses another Library, NetworkX, to store the all of these informations.
To make the maps more clear, in the examples that we will see in this section, we will simplify the map view to hide the nodes that are not interesctions or the beginning or end of a road. You can see the difference in the images below.



Unsimplified representation of a street network in Hollywood, CA.



Simplified representation of a street network in Hollywood, CA.

3.4.2. Searching for Paths

Searching for a path between 2 vertices of a network is not much more complicated than searching for a path between 2 vertices in a graph. However, in this section we will assume that we will start searching for a path by having the ids of the 2 nodes, otherwise if we only had the names of two places, it would be necessary to convert the names to coordinates and then look for the nearest vertices to these in our graph.

The OSMnx library makes it really easy to find the shortest path between 2 points with a built-in method, but we are not going to use it. We will instead use the Dijkstra's and the A* algorithms that we analyzed above, with some slight changes to make them interact with the network returned from OSMnx and to return the number of vertices and edges explored by each algorithm. Moreover, we will analyze both algorithm in 2 different scenarios, finding the shortes and the fastest path, before comparing their performances.

Since the code for each one of these algorithms is rather long and cannot run natively on a jupyter notebook, there will be a link to a GitHub Gist with the code and some instructions to run it for each of the algorithms.

All the algorithms will share some similar code, indeed the only code that will change will be the `Graph` class. The common code contains:

- A `PriorityQueue` class, that we have already seen while analyzing the Dijkstra's and A* algorithms;
- The configuration for the OSMnx library;
- 2 calls to the `geocode` method, to get the coordinates of 2 points;
- Some code the determine the boundaries of the graph that we are requiring based on the coordinates of the 2 points;
- A call to the `graph_from_bbox` method, to request the graph (or get if from the caches if already downloaded);
- 2 calls to the `get_nearest_node` method, to get the nearest nodes to the 2 points;
- The initialization of the `Graph` Class;

- A call to either the `dijkstras` or `astar` method of the `Graph` class, to compute the shortest path;
- A `print` statement to show in the console the how many vertices and edges have been explored by the called method;
- A call to the `plot_graph_route` method to plot the graph downloaded by the library with the route computed by on eof my algorithms.

3.4.2.1. Shortest Path

Finding the shortest path means using the distance between the vertices as the cost to get there. Because the vertices store the distance between them and their neighbors, retrieving the cose is really simple and does not require additional operations.

With Dijkstra's Algorithm

> Code on GitHub

We can see from the code that the algorithm is really similar to the one we analyzed before. The only thing that differs is the way the neighbors and the cost to get to a vertex are retrieved, but only because it is working with a different kind of graph. This algorithm is also the one used by default by NetworkX, and for extent by OSMnx, to calculate the shortest path on weighted graphs, as we can see on the library's source code.

Because the only 2 operations that changed (getting the neighbors and determining the cose to reach it) require the same time complexity as the the operations in the example of Dijkstra's algorithm that we analyzed before, the time complexity of this algorithm will also be $O(|V| + |E| \cdot log(|V|))$.

With A* Algorithm

> Code on GitHub

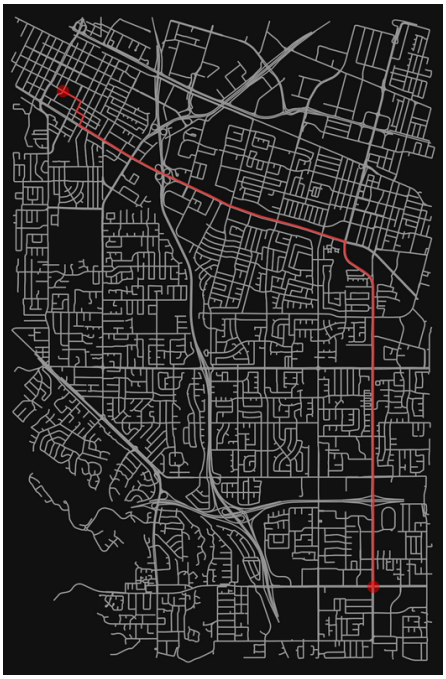
We can see that the code of this algorithm is also really similar to the one we analyzed before, but the difference here is not only in the different way to get the neighbors and their cost, but also in the heuristic function. The heuristic in this case also calculates the distance, but not in meters instead of coordinates units. To do this, we use the Haversine Formula.

The only 2 operation that changed here are the same that changed in Dijkstra's algorithm, but we have seen that the time complexity of this algorithm depends on its heuristic funciton. Like we saw while analyzing this algorithm, its worst-case time complexity remains $O(b^d)$.

Results

If we run both scripts, the paths found by the two algorithms on an example route from Cupertino, CA to Mountain View, CA, will be the following:

The path found by Dijkstra's algorithm.



The path found by the A* algorithm.

The path found by the two algorithms is the same but let's take a look at the output of both algorithms in the terminal:

```
Dijkstra's (Distance): Explored 3867 vertices and 9359 edges. The destination is 10443.014 meters away
```

Terminal Algorithm of Dijkstra's algorithm.

```
A* (Distance): Explored 1676 vertices and 4180 edges. The destination is 10443.014 meters away
```

Terminal Output of the A* algorithm.

We can see that the A *Algorithms explored 56.58% less vertices and 55.33% less edges. We can see here that the A* algorithm can really be more efficient than Dijkstra's, but still this is not the best it can do.

3.4.2.2. Fastest Path

To find the shortest path we consider the time needed to travel between two vertices as the cost to get to a vertex. However, the time needed to reach an adjacent vertex is not among the properties of our nodes. We therefore need to calculate it using the `distance` property, which can be easily retrieved, and either the `maxspeed` property (for a more precise result) or the `highway` property, which returns the road type and we can estimate the maximum speed allowed on that using some default values. If for some reason both properties are not present, we will use a default value. Once we have the distance and the speed, we can easily calculate the time.

Both of the algorithms in this case will store the default speed for different types of road and will have 3 additional methods: `get_speed`, `is_mph`, and `to_ms`. These methods determine the speed of the current road, check if the speed is in miles per hour and convert the speed to meters/second respectively.

With Dijkstra's Algorithm

> Code on GitHub

We can see from the code that apart from the new methods and the different way to calculate the cost, this algorithm is also basically the same as the one we analyzed before.

What changes here is the way the cost to get to an adjacent vertex is calculated (using the 3 methods), but since these mathods all have a worst-case time complexity of $O(1)$, the overall time complexity of this algorithm will always be $O(|V| + |E| \cdot log(|V|))$.

With A* Algorithm

> Code on GitHub

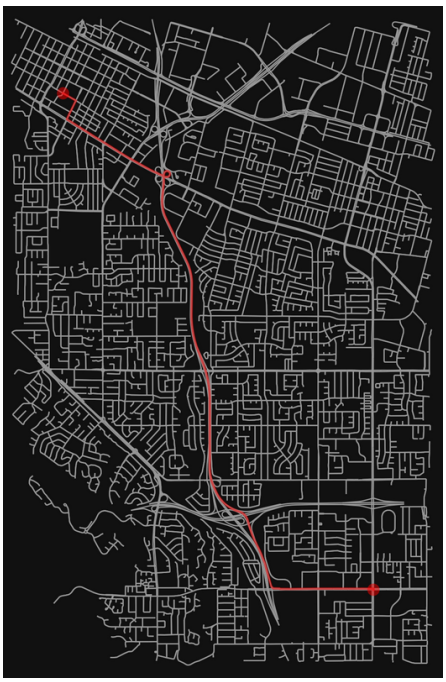
This algorithm differs from his shortest-path counterpart because of the methods needed to retirieve and convert the speed, the different way in which the cost to get to a vertex is calculated, and for the heuristic function. The heuristic function in this algorithm also calculates the distance between 2 points using the haversine function, but it also divides that value by the highest possible value of `maxspeed` so that the estimated cost is always an underestimate. In this way, we make sure that the path is always optimal.

Like we saw before, the time complexity of the A* algorithm depends on its heuristic and cannot be worst than $O(b^d)$.

Results

By running both scripts, the paths found by the two algorithms on the same example route from Cupertino, CA to Mountain View, CA, will be the following:

The path found by Dijkstra's algorithm.



The path found by the A* algorithm.

We can see that in this case the fastest path is different from the shortest path and that, even in this case, both algorithms have found the same path.

```
Dijkstra's (Time): Explored 3709 vertices and 9001 edges. The destination is 529.6689996227444 seconds away
```

Terminal Algorithm of Dijkstra's algorithm.

```
A* (Time): Explored 2977 vertices and 7298 edges. The destination is 529.6689996227444 seconds away
```

Terminal Output of the A* algorithm.

By looking at the terminal ouput of both algorithms, however, we can see a smaller difference. The A* Algorithm explored 19.74% less vertices and 18.93% less edges while still finding the same path.
This heuristic function can be improved in different ways, but it is important that its time complexity remains constant. If we really care about the performance of our algorithm but not about its optimality, we could use an heuristic function that does not guarantee optimality (in this case the fastest path) but explores less vertices. If we know the characteristics of the street network, or of a graph in general, we could create an heuristic that prioritizes some paths over others. In this case, we may want to prioritize faster roads, so what I came up with is an heuristic that looks like this:

$$h(currentV, finalV, speed) = \frac{haversine(currentV, finalV)}{maxSpeed} * \frac{thresholdSpeed}{speed}$$

Basically this heuristic function calculates how much time it would take to get from the current vertex to the final vertex, and multiply this value by a treshold speed divided by the current speed. In this way, if we set our treshold to $25m/s$ $(90km/h)$, all the vertices in roads with a higher max speed will have an even lower priority (which means that they will explored before) because we will multiply the estimate by a number which is less than 1. For all the roads with a max speed less than the treshold, the priority will be higher (and so they will be explored later). This heuristic could lead to overestimates in many different situations, but especially to find two vertices that are really far away, this could make the algorithm really more efficient. Let's try to apply this heuristic to our previous scenario to see what happens:

```
A* (Time, Non-Optimal): Explored 545 vertices and 1222 edges. The destination is 529.6689996227444 seconds away
```

Terminal Output of the Non-Optimal A* algorithm.

As we can see, in this case we still get the optimal result and compared to Dijkstra's algorithm it has explored 85.31% less vertices and 86.43% less edges. If we want to run our algorithm on very large maps, it could also make sense to dynamically choose between the two heuristics based on the distance of a vertex from the destination one.

3.4.2.3. Acknowledgemnts on the Examples

Some info about the example scenario: the map used in both cases has an area of ~60km^2 and contains 3791 vertices. The fastest path contains 45 vertices while the shortest 55.

Using this info, another way to determine how well the algorithms did could be comparing the number of explored vertices to the number of vertices in the path. We could divide the number of explored vertices by the number of vertices in the path, and the closest the number is to 1.0, the more efficient is the algorithm. Alternatively, we could also divide the number of explored vertices by the numebr of total vertices in the map, In this case, the lower the number, the more efficient the algorithm.

The efficiency comparisons are done on the same map but these Algorithms have been tested on a limited amount of maps, for this reason, the performance indicated here is not of scientific precision.

The default values for the speed of the road should be different based on the traffic laws of the country/state/administrative region in which the map is located to get more precise estimates of the travel time.

The heuristic function of the A* algorithm for the fastest path should use the maximum speed limit available among the roads present in the map to be even more efficient.

Sources

- Skiena, S. The Algorithm Design Manual. 1998. Springer.
- Udacity - Data Structures & Algorithms in Python
- Cambridge Dictionary - Algorithm
- GeeksforGeeks - Space Complexity
- Khan Academy - Asymptotic Notation
- The Lean Blogs - Some common runtime complexities and their meanings
- GeeksforGeeks - Design Techniques
- Geeksforgeeks - Backtracking
- Programiz - Master Theorem
- GeeksforGeeks - Array
- GeeksforGeeks - Linked Lists
- GeeksforGeeks - Doubly Linked Lists
- GeeksforGeeks - Stacks
- GeeksforGeeks - Queues
- CS Dojo - Hash Tables
- Ananda Gunawardena - CMU - Hash Table Conflict Resolution
- Typeocaml - Height, Depth and Level of a Tree
- GeeksforGeeks - Red-Black Trees
- GeeksforGeeks - Ropes
- Opengenus - Ropes
- GeeksforGeeks - Heaps
- HackerRank - Heaps
- Tutorialspoint - Graphs
- GeeksforGeeks - Types of Graphs
- BigO complexities [pdf]
- GeeksforGeeks - Searching Algorithms
- CS Dojo - QuickSort
- DeepAI - QuickSort
- StudyTonight - MergeSort
- GeeksforGeeks - HeapSort
- University of Maryland - HeapSort Analysis

- Back to Back SWE - Counting Sort
- Edd Mann - DFS and BFS
- GeeksforGeeks - Dijkstra's Algorithm
- Stack Abuse - A* Algorithm
- University of British Columbia - A*
- Boeing, G. 2017. “OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks.” Computers, Environment and Urban Systems. 65, 126-139. doi:10.1016/j.compenvurbsys.2017.05.004
- RosettaCode - The Haversine Formula