HSYLC 2020: Introduction to Data Structures and Algorithms

Max Guo

Summer 2020

Contents

1 Day 1

1.1 Overview of Algorithms

What is an algorithm? The modern tech world uses this term all the time, but formalizing the concept can require some thought. Intuitively, an algorithm is any procedure that tells you how to do something. As an example, a cooking recipe is an algorithm! The recipe tells you, step by step, how to turn cooking ingredients into a delicious meal. In a more complicated scenario, an algorithm may tell you the steps to get from your current location to a different place you want to go. Your favorite map app uses these types of algorithms all the time to get you places quickly and efficiently.

Actually, the formal definition of an algorithm isn't so different from what we've just described. According to Google, an algorithm is "a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer." As mentioned in the end, the study of algorithms form a large sector of computer science, although they are certainly also useful in other settings!

Discussion. Think of some algorithms that we use in our everyday lives.

As you can imagine, there are tons of different algorithms for different situations, accomplishing different tasks. We can't possibly study each algorithm in detail, but we can ask questions that apply to all algorithms. One of the most common questions to ask is how many resources does the algorithm require? This question is important to all algorithm developers - most of the time, people would prefer algorithms that consume fewer resources to those that consume more. In the cooking example, you might think of the algorithm as requiring ingredients, different containers and cookware, the time needed to prepare the food, etc. Given the same ingredients and the same meal (end result), an algorithm that takes the least amount of resources sounds very appealing!

In computer science, there are a few resources that are of primary concern. The first is the idea of "time." Algorithms that take less time are generally considered to be better (makes sense, right?). Another resource is typically called "space." Intuitively, given that our computers all have finite storage, algorithms that use less storage are generally preferred to those that use more. In the next section, we will begin to formalize these ideas.

1.2 Asymptotic notation

Let's start by considering again the resource of time. Usually, we think about time in terms of the units that us, humanity, have defined. These units include seconds, minutes, hours, days, months, and so on. However, we run into a little bit of a problem when we try to extend these ideas to testing the speed of an algorithm. For example, the time it takes to run some computer program will often depend on whether we run the program on a high speed computer or a low speed one. To abstract away these concerns, we often think of the running time of an algorithm as a function of its input size. Continuing the example of a cookbook recipe, it intuitively makes sense that the more ingredients you have, the more time it will take to prepare the food. In other words, as the "input size" (number of ingredients) grows, the amount of time required by the algorithm grows. In computer science, we can describe a relationship like this with a mathematical function.

It may take a bit of time to get used to, but we often denote input size as an integer n. We then say that the time it takes for our algorithm to run will then be a function of n. It's hard to precisely describe what the function will be, because that depends on your definition of 1 step. Is an algorithm for writing down

n numbers n steps, one for each number? Or perhaps $n + 1$ steps, because we have to first put our hand on the paper? Or maybe some even more complicated function? We also want to abstract away these concerns, motivating the concept of *asymptotic notation*.

Asymptotic notation will allow us to stop caring about whether or not an action takes "7 time" or "1 time", as both perspectives will result in the same asymptotics. Moreover, asymptotic notation will give us a glimpse into the limiting behavior of algorithms, or how they behave as the input sizes get larger and larger. As computers can process pretty large numbers, these abstractions will align well with the current, physical technology.

Let's dive in - this part will get a little more mathematical and technical.

Definition 1 (Big-Theta)**.** For functions $g(n)$ and $f(n)$, we say that $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2, N > 0$ such that for all $n \geq N$, we have that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.

A few very similar concepts:

Definition 2 (Big-O)**.** For functions $g(n)$ and $f(n)$, we say that $f(n)$ is $O(g(n))$ if there exist constants $c, N > 0$ such that for all $n \geq N$, we have that $0 \leq f(n) \leq cg(n)$.

Definition 3 (Big-Omega)**.** For functions $g(n)$ and $f(n)$, we say that $f(n)$ is $\Omega(g(n))$ if there exist constants $c, N > 0$ such that for all $n \geq N$, we have that $0 \leq cg(n) \leq f(n)$.

We will usually stick to big-O notation, because we often want to know the worst case scenarios for our runtimes. Let's give a few examples for intuition:

- n is $O(n)$.
- $2n + 7$ is $O(n)$.
- $100000n + \sqrt{n}$ is $O(n)$.
- n^2 is NOT $O(n)$. Note that we have $n^2 > n$ for all $n \geq 2$, so the definition doesn't satisfy! Think about this for a little bit.

This means that we do not have to care about the constant factors - the function $f(n) = n$ is $O(n)$, but so is $f(n) = 100n$! This is a little bit unrealistic in real life, because in practical matters constant factors often do matter. For the most part, however, constant factors are small enough that asymptotics dictate the growth of a function.

There's also a little bit of a technical issue here - from the definition alone, we also have true statements like $100n$ is $O(n^2)$. This is technically true, but we often prefer to write the tightest bound that we can put on the function inside the O (If you guys remember big Theta, this actually becomes a slight abuse of terminology, since we are using big-O as big-Theta).

1.3 Induction

As computer scientists, we must be completely sure about all the claims we make, as careless mistakes can sometimes lead to huge consequences. Mathematics gives us the tools for this. In this next section, we will cover a technique that will allow us to prove statements we make in the future, including the correctness of certain algorithms and more.

I included an article on induction, but it's fine if you didn't read or understand it yet. Essentially, for our purposes, induction is a mathematical proof technique that allows us to prove the correctness of some statement for all positive integers. We first prove that a statement is true for one particular integer (base case). We then prove that, if a statement is true for some integer, it is true for the next integer (inductive step). This then rigorously shows that the statement is true for all integers greater than or equal to the base case.

Let's try this on a simple example, though perhaps not as simple as in the reading. Let's show that the sum of the first n odd integers is n^2. First of all, let's intuitively check that this is true: $1 = 1$. $1 + 3 = 4 = 2^2$. $1 + 3 + 5 = 9 = 3^2$. Seems true enough.

Now let's prove this by induction. The base case is satisfied, as $1 = 1$ indeed. Now let's assume this statement is true for the first k odd integers, and let's try to prove the statement for the first $k + 1$ odd integers:

$$\begin{aligned} 1 + 3 + \cdots + (2k-1) &= k^2 \\ \implies 1 + 3 + \cdots + (2k-1) + (2k+1) &= k^2 + 2k + 1 \\ \implies 1 + 3 + \cdots + (2k+1) &= (k+1)^2, \end{aligned}$$

which completes the proof.

Now let's move on to the inductive proofs with big-O notation. As an example, let us show that $n \in O(2^n)$. We want to show that there exist constants $c, N > 0$ such that $n \leq c2^n$ for all $n \geq N$. Take $N = 1$ and $c = 1$. Our base case is $n = N = 1$. We can readily verify that $1 \leq 2$. Now assume this holds true for $n = k$, so that $k \leq 2^k$. We wish to prove that $k + 1 \leq 2^{k+1}$. Let us simplify from what we know:

$$\begin{aligned} k &\leq 2^k \\ k + 1 &\leq 2k \\ 2k &\leq 2^{k+1} \\ \implies k + 1 &\leq 2^{k+1}, \end{aligned}$$

which proves the inductive step. This was a relatively simple example, but it rigorously proved what we intuitively knew.

Example 4. Given the Fibonacci numbers are defined

$$F_0 = 0, F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2,$$

Prove that $\sum_{j=0}^{n} F_j = F_{n+2} - 1$.

Activity. I will assign each person a different function, and I want everybody to work together to find the correct order, from biggest to smallest, for big-O notation. Keep in mind that two functions may be the same in terms of big-O notation!

n^2	$2n^2 + \log n$
5	$10000n$

2 Day 2

2.1 Review and Extension

Yesterday we introduced what an algorithm was and a few tools that we'll use to analyze algorithms: asymptotic notation and induction. Before we get into the new material today let's give a few facts relevant to yesterday's material:

Theorem 5. *If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(g_1(n) + g_2(n))$.*

Proof. By the definition of big-O, we have that there exist positive constants c_1, c_2, N_1, and N_2 such that $f_1(n) \le c_1 g_1(N)$ for $n \ge N_1$ and $f_2(n) \le c_2 g_2(n)$ for $n \ge N_2$. Take c to be $\max(c_1, c_2)$ and N to be $\max(N_1, N_2)$. Then $f_1(n) + f_2(n) \le c(g_1(n) + g_2(n))$ for all $n \ge N$, which is what we wanted to prove. $\square$

Example 6. Given the Fibonacci numbers are defined

$$F_0 = 0, F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \ge 2,$$

Prove that $\sum_{j=0}^{n} F_j^2 = F_n F_{n+1}$.

2.2 Introduction to Data Structures

Without getting too philosophical, I will assume we all know what data is. A data structure is, just like it sounds, something that organizes data in some useful way. Another way to phrase it would be a special way of organizing and storing data in a computer so that it can be used efficiently. In computer science, we have some simple data structures that we'll be taking a look at today.

A simple example of a data structure is a list. A list is an ordered sequence of arbitrary elements. In some computer programming languages, each element is required to be of the same type (for example, integers, real numbers, characters, and strings are often considered different types). However, this is not the case in Python. Here's an example of what a list looks like in Python:

```python
L = [5, 3, 2, "Hello World", 5.2]
```

We will denote a general list as $[x_1, x_2, \ldots, x_n]$. Then n is called the *length* of the list, x_1 is the *head* of the list, and x_n is called the *tail* of the list. The list L is indexed by the values you see in the subscripts: the first value of L is denoted by $L[1]$, which is x_1, the second value of L is denoted by $L[2]$, which is x_2, $\ldots$, $L[n]$ is the last value in L, x_n.

Note that if there is some comparison operator between all the values (for example, suppose all elements of the list are integers). Then we have a notion of whether or not a list is sorted.

Indexing can actually be very confusing in the beginning, since Python and most other languages actually index starting with 0, so the first element is $L[0]$ and the last element is $L[n-1]$.

There are also common operations on lists that we are often interested in:

- isEmpty: returns True if our list is empty and False if not.
- Push Back: puts another element x in the back of the list.
$$[x_1, x_2, \dots, x_n] \to [x_1, x_2, \dots, x_n, x]$$
- Push Front: puts another element x in the front of the list.
$$[x_1, x_2, \dots, x_n] \to [x, x_1, x_2, \dots, x_n]$$
- Pop Back: removes an element from the back of the list.
$$[x_1, x_2, \dots, x_n] \to [x, x_1, x_2, \dots, x_{n-1}]$$
- Pop Front: removes an element from the front of the list.
$$[x_1, x_2, \dots, x_n] \to [x_2, \dots, x_n]$$

Note that the naming you see here is not super consistent throughout the field - there are other names for these operations, as you'll soon see. Lists are also super dynamic in Python - they can shrink and expand and be manipulated with ease. We show the Python equivalent of these operations in a Jupyter Notebook.

Certain data structures that specifically support a subset of the operations that a list supports are also very useful:

- A *stack* implements isEmpty, push front, and pop front.
- A *queue* implements isEmpty, push back, and pop front.
- A *deque* implements all of these operations.

I often imagine a stack as a bunch of plates stacked on top of one another. The operations you can do are check if there any plates, take a plate off the top, or put another plate on top. It follows the paradigm LIFO: the last element in is the first element out. For queues, you can imagine a bunch of people waiting in a line to buy movie tickets. You can put someone into the back of the line, or check if someone is in line, or remove the person in the front of the line (presumably giving them a ticket). Queues follow the paradigm FIFO: the first element in is the last element out. A deque is short for "double ended queue", and you can do whatever you want on either end of the list.

A small note: "implement the operation" this whole time usually meant that these data structures are able to perform their respective operations in $O(1)$ (constant) time.

We haven't actually mentioned what data structure the Python list falls under. Python lists implement, in $O(1)$ time: push back, pop back, element access and update, and length. It is an example of a *dynamic array*. Note that stacks, queues, and deques do not a-priori implement some of these, like element access and length!

2.3 Searching

The problem of searching a list is very practical and a good example of different algorithms. Informally, given a list L and some value x, we want to "find x" in L or raise some sort of error / return some special value if x is not in L. "find x" can mean find the index of x in L, or just return True if x is found. Before we get into the nitty-gritty, we're going to play a game to get us acquainted with this problem!

Activity. Pair up with someone and play this game. Person 1 thinks of a number in their head, between 1 and 100 (inclusive). Person 2 repeatedly tries to guess this number. Every time Person 2 guesses incorrectly, Person 1 tells Person 2 whether their number is higher or lower. Keep track of the number of guesses - after switching roles, the person with the least number of guesses wins!

One way to search a list is to just go from the beginning to the end and check, at each index, whether or not the element at that index is equal to x. This can be implemented as follows:

```python
L = [5, 4, 3, 2, 1]
print(f"Our list is {L}")
x = 5
found = False
for i in range(len(L)):
    if L[i] == x:
        print(f"{x} is located at index {i}")
        found = True
if not found:
    print(f"{x} is not located in the list")
```

This algorithm takes $O(n)$ time to run, where recall that our list size is n. Now, this is sometimes the fastest you can do! Given an arbitrary list with no structure, you have to go through each element and check to see if that element matches x.

If we assume that the list is sorted (let's just suppose the list has all numbers for now), then as you may have already guessed from our game, we can do much faster! We can keep comparing our value to the middle of the current list, then we can essentially discard half our list until we've either found the element or determined it doesn't exist. This algorithm is called *binary search*.

A formal implementation of binary search in Python is the following:

```python
L = [1, 2, 54, 9, 8, 22, 5, 14, 0, 31, 43]
L.sort()
print(f"Sorted list L: {L}")
x = 5

# Binary search to find x in L
left = 0
right = len(L) - 1
found = False
while(left <= right):
    mid = (left + right)//2
    if L[mid] == x:
        print(f"Found {x} at location {mid}!")
        found = True
        break
    elif L[mid] < x:
        left = mid + 1
    else:
        right = mid - 1

if not found:
    print(f"{x} not in L")
```

How long does binary search take? If our list originally had n elements, we essentially reduce it down to $n/2$ elements after 1 search, then $n/4$ elements after 2 searches, etc. In other words, in the worst case scenario of having to search until we are left with a list of 1 element, the number of steps we use is the number of times you can divide n by 2 before getting 1. This is $\approx \log n$ operations, so we say that binary search takes $O(\log n)$ time. This is much faster than linear search, $O(n)$ time, assuming that our list is sorted!

3 Day 3

3.1 Review and Extension

Let's talk about a very useful application of Binary Search, which is finding the first integer such that some function of the integer returns true (or false), or the last integer such that some function of the integer returns false (or true). These problems are very related, and the first can be pictorially represented as finding x in the following picture:

1	2	...	$x-1$	x	$x+1$	...	n
False	False	...	False	True	True	...	True

Similarly, finding the last integer such that a function returns false would respectively choose $x-1$ in the above picture.

Suppose we are trying to find the first integer such that a function returns True. This can be solved in binary search in the following way: test the midpoint integer m. If it is True, then repeat on the range $1 \dots m$. If it is False, then repeat on the range $m+1 \dots n$. Continue until the left end of the range is $\geq$ the right. If left is equal to $n+1$, there was no True to begin with! Otherwise we have found x. The Python code is below:

```
L = [False]*50 + [True]*20 # First True should be at L[50]

left = 0
right = len(L)
while(left < right):
    mid = (left+right)//2
    if L[mid]:
        right = mid
    else:
        left = mid+1
if left == len(L):
    print("L has all False's")
else:
    print(f"L's first True appears at {left}")
```

We can do a very similar approach for the other problem of finding the last False - notice how it would simply be the answer returned by the above algorithm, minus 1.

3.2 Sorting!

We've alluded to the problem of sorting in the past two days, and today we'll be covering it in detail! The problem is simple: given a list, what is an time and space efficient method of sorting the list?

There are many different algorithms out there that solve this problem, and we'll go over some of the simpler and more instructive ones.

3.3 Insertion Sort

Insertion sort is all about maintaining a sorted subarray of the original array by taking each element and sorting it into the right place.

Example. [5, 1, 6, 4, 3]

Verbose Explanation. Suppose we start with the array: [5, 1, 6, 4, 3]. Insertion sort starts from left to right and ensures at iteration i that the first i elements are sorted. In the first iteration, [5] is indeed sorted, so there is nothing to be done. Our list remains at [5, 1, 6, 4, 3]. In the second iteration, we start at the second number 1, and look at the preceding element 5. Since $5 > 1$, we swap 1 with 5 and obtain [1, 5, 6, 4, 3] as our list. In the third iteration, we start at the third element 6. $6 > 5$, so nothing needs to be done in this iteration. In the fourth iteration, we start at the element 4. Since $6 > 4$, we swap the elements 6 and 4 and obtain [1, 5, 4, 6, 3]. We continue to look at the element 4, and since $5 > 4$, we then swap 4 and with 5 and obtain [1, 4, 5, 6, 3]. Now since $4 > 1$, we don't swap 4 and 1, and we are finished with the fourth iteration. The final iteration of insertion sort: We take our value 3, and since $6 > 3$, we swap 3 and 6. Then, since $5 > 3$, we swap 3 and 5. Since $4 > 3$, we swap 3 and 4. Finally, since $3 > 1$, we do not perform any more swaps. This was our last iteration, and insertion sort terminates. Our final list is [1, 3, 4, 5, 6], as desired.

Python code for Insertion Sort is given below:

```
L = [5, 1, 6, 4, 3]

for i in range(len(L)):
    curr = i
    while( curr > 0 and L[curr] < L[curr-1] ):
        L[curr], L[curr-1] = L[curr-1], L[curr]
        curr -= 1
```

```
print(L)
```

Analysis. The code allows us to analyze the worst-case runtime of this algorithm. For element in position i in L, we can do up to i swaps. In other words, the number of operations we do is approximately $0+1+2+\cdots+n-1$ (n is the length of the list). This algorithm is roughly $O(n^2)$.

3.4 Merge Sort

Merge sort is another common sorting algorithm that relies on the "divide and conquer" paradigm. In other words, we'll try splitting our list into two halves (roughly), sorting each one of them, then combining then back into a sorted list. This is known as a recursive algorithm.

Example. [1, 4, 2, 7, 6, 3, 5, 8].

Python code for Merge Sort is given below:

(Code taken from GeeksforGeeks)

```
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2 # Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+= 1
            else:
                arr[k] = R[j]
                j+= 1
            k+= 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i+= 1
            k+= 1

        while j < len(R):
            arr[k] = R[j]
            j+= 1
            k+= 1

L = [1, 4, 2, 7, 6, 3, 5, 8]
print(L)
mergeSort(L)
print(L)
```

Analysis. Mergesort is a bit harder to analyze than insertion sort, and we'll do it a bit differently. We know the runtime of Mergesort is some function of n, so let's call it $T(n)$ (T is commonly used in these runtime scenarios, as opposed to other common function names like $f(n)$). Inside mergesort, we call mergesort twice, so we have that:

$$T(n) = 2T(n/2) + \text{some more stuff.}$$

Now, the other stuff that we need to do involves merging the two sorted halves. We iterate through each of the elements once, so this is about $O(n)$ work. This means that our equation is:

$$T(n) = 2T(n/2) + O(n).$$

The above equation is known as a *recursive equation*, or a *recursion*, which is our next topic!

3.5 Recursions

I claim that the answer to the above equation is $O(n \log n)$. This can be proved formally via induction and using the definition of asymptotics, but we won't go through the detail here. Instead, let's give some informal intuition for the runtime through a recursion tree.

Example: Recursion Tree.

There are some complicated recursions out there, and using a recursion tree doesn't always solve the recursion immediately. Instead, we often result to a very powerful mathematical result known as the *Master theorem* (a very grand name).

Theorem 7 (Master Theorem)**.** *Suppose we have the recursive equation:*

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k.$$

We have the following cases:

$$\begin{cases} \text{if } a < b^k & \text{then } T(n) = O(n^k). \\ \text{if } a > b^k & \text{then } T(n) = O(n^{\log_b a}). \\ \text{if } a = b^k & \text{then } T(n) = O(n^k \log n). \end{cases}$$

Looking at the master theorem, we see that our recurrence is in the case where $a = 2$, $b = 2$, and $k = 1$, so we have the third case, or $T(n) = O(n \log n)$, as desired.

4 Day 4

Today will be a lot of new definitions, especially if you haven't seen graphs before. Don't worry though, we'll provide many examples! Let's dive right in:

4.1 Graphs

Definition 8 (Graph)**.** A graph is a set of vertices (also called nodes) and edges, where edges are pairs of vertices. We say an edge is directed if there is a starting vertex and an ending vertex, an an edge is undirected if there is no notion of beginning or end. A graph can also be weighted if edges have a number associated to them.

Here is a nice visual representation of a directed (unweighted) graph:

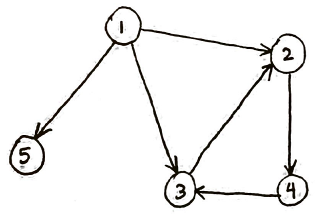


Figure 1: A directed graph with five vertices and six edges

Graphs can be used to represent many things in real life. For example, consider a graph of relationships. The nodes in this graph might represent people, and the edges would represent whether or not two people know each other. Another weighted graph might represent flights between major cities, where the nodes would be cities, the edges would be flights, and the weights would represent the cost of flights between two cities. These are just a few examples!

In the previous example, notice how you can walk from 1 to 4 in a few different ways: $1 \to 2 \to 4$, or $1 \to 3 \to 2 \to 4$. These are called *paths.*

Definition 9 (Path)**.** A path in a graph is a finite sequence of edges which joins a sequence of vertices which are all distinct (and since the vertices are distinct, so are the edges).

Notice also how we can "walk" from $2 \to 4 \to 3 \to 2$. Since we start and end at the same vertex, this is known as a *cycle.*

Definition 10 (Cycle)**.** A cycle is a sequence of edges which joins a sequence of vertices that starts and ends at the same vertex. A graph that does not contain any cycles is called an acyclic graph.

Finally, the different ways of drawing graphs are definitely not unique! The following graph is equivalent to the above graph (verify this!):

A few more definitions and examples:

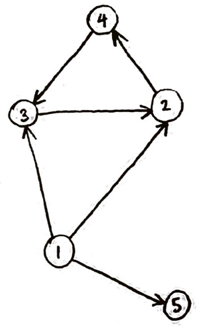


Figure 2: The same graph!

Definition 11 (Connectedness)**.** In an undirected graph, two vertices are connected if the graph contains a path between the two.

An undirected graph is connected if every pair of vertices in the graph is connected.

Definition 12 (Trees and Forests)**.** A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, a tree is a connected acyclic undirected graph.

A forest is an undirected graph in which any two vertices are connected by at most one path. A forest need not be connected, but it is the disjoint union of trees.

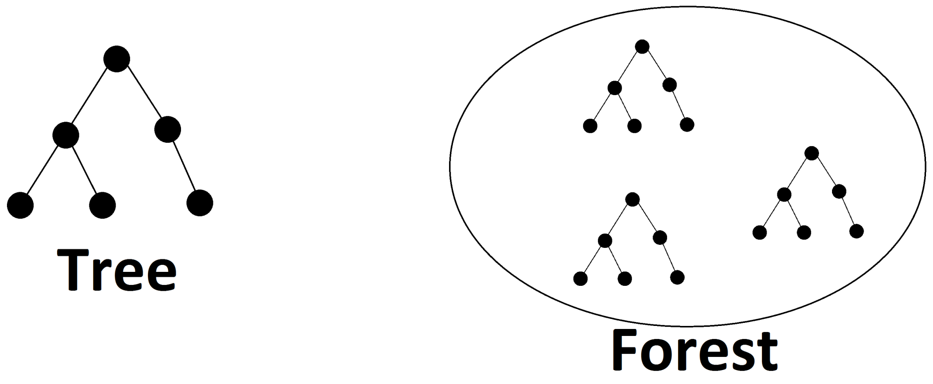


Figure 3: Trees versus Forests

4.2 Representing Graphs

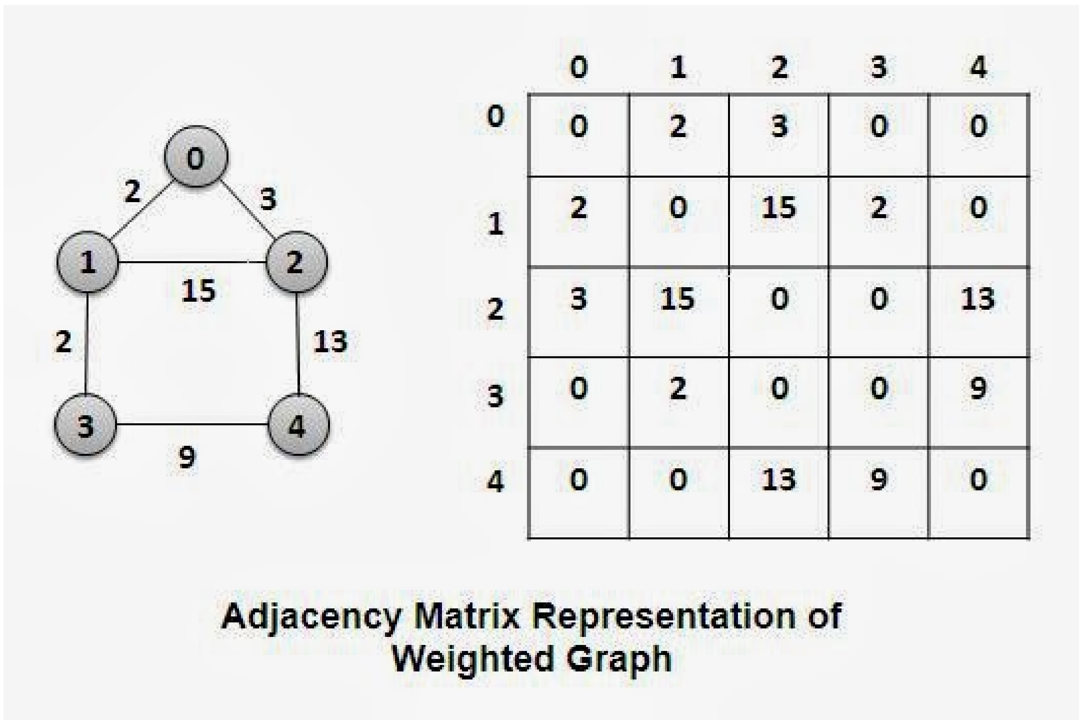
We now have a little intuition as to what a graph is. However, these structures may still seem a little abstract - how do computers concretely remember graphs? There are two major ways of representing graphs, by adjacency matrices and adjacency lists.

4.2.1 Adjacency Matrix

Suppose we have an unweighted, directed graph whose nodes are labeled from 1 to n. Then, an adjacency matrix is an $n \times n$ matrix of zeroes and ones, where there is a 1 in position (i, j) if and only if there is an edge going from node i to node j.

In the case of an undirected graph, there is a 1 in positions (i, j) and (j, i) if and only if there is an edge between nodes i and j.

In the case of a weighted graph, we replace the 1 with the weight. Take a look at Figure 4.



	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Figure 4: Adjacency matrix for weighted, undirected graph.

4.2.2 Adjacency List

Another way of representing graphs is to put the nodes that a certain node i is connected to into a list. We then have a list of lists (a list for each node). The advantage of this method is that we don't have to necessarily store n^2 values when there aren't that many edges in the graph (when the graph is *sparse*). We can include weights in the information if the graph is weighted.

4.3 Depth-First Search (DFS)

Depth first search will allow us to answer many interesting questions we might want to know about graphs. For example, how can we efficiently tell whether or not a graph has a cycle? How can we tell if a graph is connected? We will soon see how to answer these questions with DFS.

DFS keeps track of what nodes have been "visited" and which ones have not. Iterating through the nodes

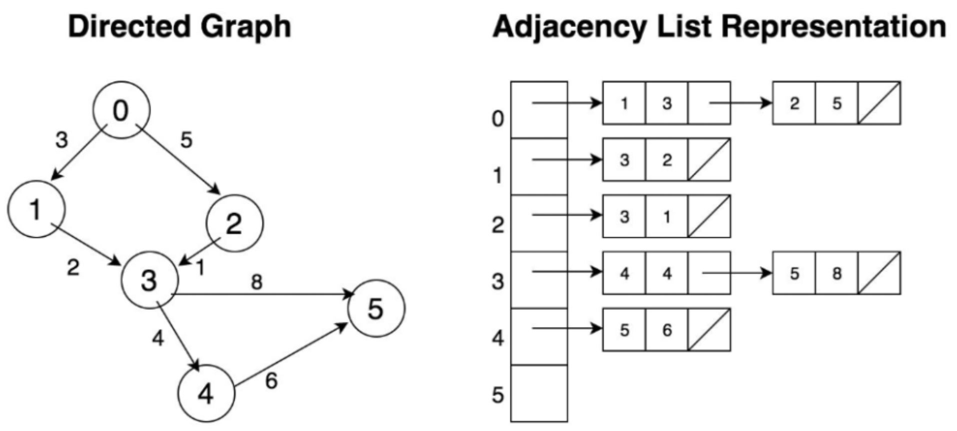


Figure 5: Adjacency List for a Directed Graph

that have not been visited, it recursively calls itself and upon encountering a node that hasn't been visited until it reaches a node that has no more un-visited neighbors, in which case it backtracks until it reaches a node that has an un-visited neighbor.

Here is Python code that performs DFS on the first graph diagram above, using an adjacency list representation:

```
adj = [
    [1, 2, 4],
    [3],
    [1],
    [2],
    []
]

visited = [False]*5

def dfs(node):
    visited[node] = True
    print(f"{node} has been visited.\n")
    for neighbor in adj[node]:
        if not visited[neighbor]:
            dfs(neighbor)
    return

for node in range(len(adj)):
    if not visited[node]:
        dfs(node)

print("DFS complete.")
```

DFS has an implicit stack structure - it keeps putting nodes in a stack and then calling itself on the top of the stack (popping the nodes in the process by marking them visited) and putting an unvisited neighbor onto the stack. Here is a visual walkthrough of the DFS we did in class:

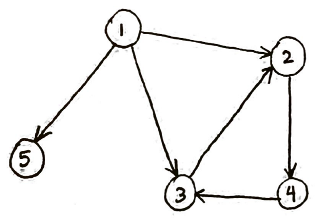


Figure 6: The original graph

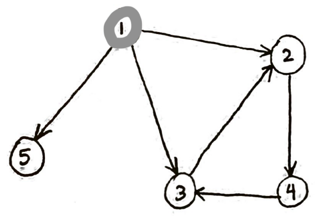


Figure 7: DFS visits vertex 1

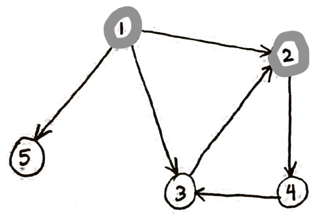


Figure 8: DFS visits vertex 2

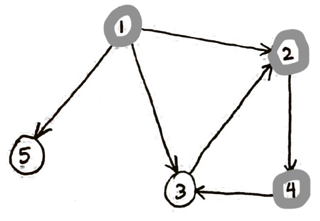


Figure 9: DFS continues down and visits vertex 4

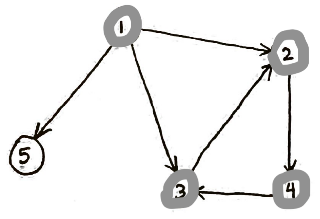


Figure 10: DFS continues and visits vertex 3

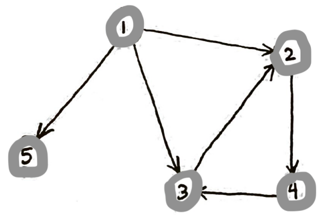


Figure 11: Vertex 2 and 3 have been visited before, so vertex 5 is visited. DFS terminates.

4.4 Topological Sort

Assuming we have a Directed Acyclic graph, we can perform a topological sort in order to obtain an ordering of a graph. In other words, topological sorting will return a list that is guaranteed to be such if node u is before node v, then there is no path from node v to node u.

How can we implement this? Well, if we assume a DAG, then once we are completely done visiting a node we can push it into a stack. We then pop the elements of the stack off one by one to obtain the correct ordering of the elements! This is because the elements finish in order, which we'll show tomorrow.

Here is a modified version of our code:

```python
# Assuming we have a DAG

adj = [
    [1, 2],
    [2, 5],
    [3],
    [],
    [],
    [3, 4],
    [1, 5]
]

marked = [False]*len(adj)

sorted_graph = []

def dfs(node):
    marked[node] = True
    print(f"Starting to visit node {node}\n")
    for neighbor in adj[node]:
        if not marked[neighbor]:
            dfs(neighbor)

    print(f"Finished visiting node {node}\n")
    sorted_graph.append(node)
    return

for node in range(len(adj)):
    if not marked[node]:
        dfs(node)

sorted_graph = sorted_graph[::-1]

print(f"Topologically sorted graph: {sorted_graph}")
```

Note that this code doesn't work if there is a cycle! We'll fix this soon!

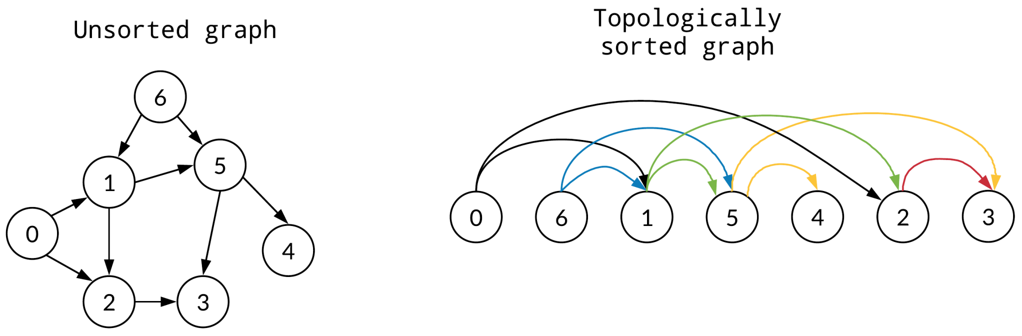


Figure 12: Topological Sort

5 Day 5

5.1 DFS Runtime

What is the runtime of DFS, in terms of V and E, the number of vertices and edges of the graph, respectively? Well, if we look at the `dfs` function, we see that each node is called with `dfs` exactly once, since it becomes visited and is thus never called again. Moreover, we see that each edge is checked once (or twice, if undirected), since the "`if not visited[neighbor]`" line in aggregate goes through all the edges. Thus our runtime is $O(V + E)$.

5.2 DFS Tree

With any DFS of a graph, we can classify the edges of the graph into four types based on the DFS. This is based on the idea of timing the search. We assign each node a start time, when we first "start" visiting it, and an end time, when we are completely done visiting it and all its neighbors. This gives us an annotated version of DFS:

```python
from tabulate import tabulate

adj = [
    [],
    [2, 3, 5],
    [4],
    [2],
    [3],
    []
]

visited = [False]*6
start_times = [0]*6
end_times = [0]*6
time = 0

def dfs(node):
    global time
    visited[node] = True
    start_times[node] = time
    time += 1
    print(f"{node} has been visited.\n")
    for neighbor in adj[node]:
        if not visited[neighbor]:
            dfs(neighbor)
    end_times[node] = time
    time += 1
    return
```

```
for node in range(1,len(adj)):
    if not visited[node]:
        dfs(node)

print("DFS complete.\n")

table = {}
table["Node"] = list(range(1,6))
table["Start Time"] = start_times[1:]
table["End Time"] = end_times[1:]

print(tabulate(table, headers=table.keys()))
```