---

title: "Algorithms and Data Structures" categories:

- University

Normally people will make entire articles out of things like Big O notation, Binary search or other algorithms. There simply is no need to blabber on about something when you can convey the same message in a shortened manner, so this is where this article comes into play.

If you're not interested in a certain algorithm (say for example, Min and Max) then simply skip it! Most of the examples given are in Python, but you need not know how Python works to understand the algorithms. Note that later in this article some algorithms may be implemented in Java.

I do not attempt to explain my code used that much. You should really try to implement this yourself in code (if you're a programmer) and perhaps use my code to check over it. My code is definitely not the *best* way to implement it, but it is a way.

By the end of this article you should have a good understanding of algorithms and data structures.

Table of Contents

1. What is an Algorithm?
2. Algorithm Complexity, Big O notation
    1. Big-O tips
    2. Big-O Summary
    3. Big-O Cheatsheet
3. Searching Algorithms
    1. Sequential Search
    2. Binary Search
4. Min and Max Algorithms
    1. Coding the Min algorithm
    2. Codign the Max algorithm
5. Datastructures
    1. Queues
    2. Stacks
    3. Linked Lists
        1. Singly Linked Lists
        2. Doubly Linked Lists
        3. Traversing Linked Lists
        4. Programming Linked Lists
            1. Adding a Node to the Head of a Linked List
            2. Deleting a Node at the Head of a Linked List
            3. Inserting an Item into a Linked List
            4. Searching a Linked List
6. Sorting Algorithms
    1. Bubble Sort
        1. Coding a Bubble Sort
    2. Selection Sort
        1. Coding a Selection Sort
    3. Insertion Sort
        1. Coding an Insertion Sort
7. Programming Linked Lists - Sorts
    1. Bubble Sort in a Linked List
    2. Programming a Bubble Sort in a Linked List
    3. Selection Sorts in a Linked List
    4. Insertion Sort in a Linked List
    5. Inserting a New Node into an already sorted linked list with Insertion Sort

What is an Algorithm?

An algorithm is a set of instructions typically undertaken by a computer to reach a targeted goal.

When you make a sandwich, you are performing an algorithm. Habits are algorithms. Do you put your socks and shoes on in the order sock > sock > shoe > shoe or do you do sock > shoe > sock shoe? Both of these are algorithms.

> Algorithms have a long history and the word can be traced back to the 9th century. At this time the Persian scientist, astronomer and mathematician Abdullah Muhammad bin Musa al-Khwarizmi, often cited as "The father of Algebra", was indirect responsible for the creation of the term "Algorithm". In the 12th century one of his books was translated into Latin, where his name was rendered in Latin as "Algorithmi".

From here

Normally algorithms break down things that we as humans think are very simple and easy into little steps that a computer can perform.

An example of this is let's say you want to find the number 3 in the list [1, 2, 3]. You look at it and you see it, but a computer can't see the number 3 in the list until it goes through every single item to check.

Computers are the fastest dumbest things to ever be created.

Algorithms are essential in Computer Science, you simply cannot live without them. Something computer scientists do alot is compare how long an algorithm takes to run against other algorithms.

How do we measure how long an algorithm takes to run?

TK have coding book tips here

We could simply run an algorithm 10,000 times and measure the average time taken but let's say we have an algorithm that took different inputs, like say for example we have an algorithm that takes a list of items and prints every item to the screen. If we only input lists of length 1 (1 item), the average time would be around 0.1 seconds. If we entered items of length 1500, the average would be different. Of course you can use some advance statistical knowledge to work out the true average by inputting lists of varying lengths or you could use Big O notation.

Big 0 notation is notation used to describe how efficient an algorithm is. It's incredibly important to know this since every major employer will question you on this and it'll most likely come up in any algorithms exams (Hello University of Liverpool people ❤️)

A hierarchy of functions exist:

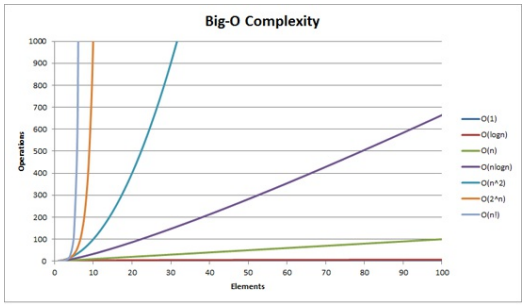
1	**log n**	**n, $$n^2$$, $$n^3$$**	**$$2^n$$**
Constant	Logarithm	Polynomial	Exponential

Where the further right they are, the longer it takes. Big O notation uses these functions to describe algorithm efficiency. O(2^n) is larger than O(1).

Log is called a logarithm (typically in base 2 binary but can differ). I shan't explain the logistics of logarithms for someone has already done it far better than I can. You do not need to watch the whole video, perhaps just the first two minutes.

https://www.youtube.com/watch?v=ZIwmZ9m0byI

A "constant" value is something that does not change depending on its input. So for example, adding 1 to 500 is the same as adding 1 to 4741838148, because it doesn't scale depending on the input size.



In Big O notation, we always use the *worst case* scenario for our calculations.

To calculate the Big O of an algorithm or selection of code, well... it's basically an educated guess. If your algorithm touches EVERY item in a list, the algorithm is 0(n). If your algorithm does not rely on input size (like an algorithm which just does 2 * n) then it is considered 0(1). If your algorithm appears to be halfing / going down alot it is *probably* log n. If your algorithm looks like it's counting up in how memory is counted (16, 32, 64, 128, 256, ..., 1024) then it is probably 2^n.

There are some super useful rules you have to obey in order to simplfy your algorithm.

Drop the constants

If you have an algorithm described as O(2n), drop the 2 so it's just O(n).

Drop the non-dominant terms

O(n^2 + n) just becomes O(n^2). Only keep the larger one in Big O.

If you have a special sum such as O(b^2 + a) you can't drop either because without knowledge of what b and a are.

Summary

Bet you were expecting some hard to understand guide to Big O huh? Well, this is all it is. You just need to memorise (or learn) the hierarchy, take some algorithms and find out what their Big O notation is.

Big O notation only represents how long an algorithm can take but sometimes we care about the memory (space complexity) of an algorithm too.

TK other forms of measuring algirhtms.

Cheatsheet

Some of these algorithms you may not know yet, but they will all be explained in this article.

Algorithm	Big O
Sequential Sort	O(n)
Binary Search	O(log n)
Minimum Value	O(n)
Maximum Value	O(n)
Quicksort	O(n^2)
Mergesort	O(n log n)
Bubble Sort	O(n^2)
Insertion Sort	O(n^2)
Selection Sort	O(n^2)

Data Structure	Access	Search	Insertion	Deletion
Array (list)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)

Seqeuntial search & Searching

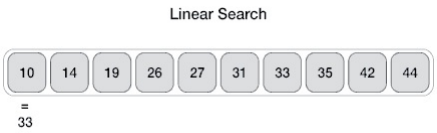
Let's say you have an array of items [1, 3, 7, 4, 9] and you want to find the number 4. You just look at it, and you see it. What's so hard about that? Well a computer doesn't have super cool abillities like us just to "see" things. It has to go through the list in order to "see" the number. There are many algorithms that allow a computer to search lists / arrays. One of them is sequential search.

A sequential search would calculate it like so:

Current number	Description
1	Not goal
2	Not goal
3	Not goal
4	Goal found

Note: this type of table is called a trace table. It shows the values and names of all variables in the algorithm every single time the algorithm / loop is run until the algorithm finishes.

Essentially you are sequentially counting up every single item in an list until you find your goal (if it exists in the list).



Note: this algorithm is sometimes called linear search.

In the *worst case scenario* the number we are looking at is either not in the list or at the very end, so in time complexity this is O(N).

```
for i in list: # for every element in the list
    if i == answer: # if you selected the element and it's the answer then
        print(answer) # print it to the screen
    else:
        continue # else continue searching
```

Now this may seem stupid, but it can work on non-sorted arrays. You use this search alot in daily life, like looking for a book or looking for an item on a menu.

Binary search

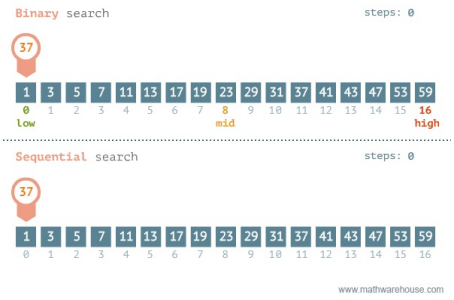
Binary search only works on sorted arrays. It basically halves the array and checks whether the halfed item is lower or higher than the goal item in the array.

Given the array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and we want to find 2 we can run binary search like this:

```
Item in middle of array = 5
is 5 less than or greater than 2? It's greater than so we throw away the right hand side (greater than) and start again.
5 / 2 = 2.5, so we round up here.
Is 3 less than or greater than 2? It's greater than so we do:
3 / 2 = 1.5
round up to 2 (as we only accecpt integers here)
2 is goal, goal is found.
```

The time complexity here is O(log n) because you're effectively asking how many times can you divide by 2 until you find the answer.

Here is a nice gif of binary search in action



This is the worst case performance for binary search, beaten by sequential search!

This code is taken from here. It's not essential to understand the Python code in order to understand the algorithm.

```
def bsearch(alist, target):
    left = 0
    right = len(alist)-1

    while True:
        mid = int((left + right)/2)
        if alist[mid] < target :
            left = mid + 1
        elif alist[mid] > target :
            right = mid + 1
        elif alist[mid] == target :
            print ("word '" + target + "' found at " + str(mid))
            break
        elif left > right:
            print ("Not Found!")
            break

i = input("Enter a search >")
alist = ["rat","cat","bat","sat","spat"]
alist.sort()
```

```
bsearch(alist, i)
```

If you're a Java programmer, have a look at this code:

```
// sortdata[] is an array in ascending order, n is size of array, key is the number we want to find
static void binary_search(int[] sortdata, int n, int key) {
                int count;
                boolean found = false;

                // start binary search on the sorted array called sortdata[]
                found = false;  // to indicate if the number is found
                count = 0;      // to count how many comparisons are made

                int temp = n - 1;

        while (found == false){
            if (sortdata[temp] == key){
                                found = true;
                        }
            else {
                if (temp < key){
                    temp = temp / 2;
                }
                else {
                    temp = temp + (temp / 2);
                                }
                        count = count + 1;

            }
        }
                System.out.print("The number " + key + " is ");
                if (found == false)
                        System.out.print("not ");
                System.out.println("found by binary search and the number of comparisons used is " + count);
        }
}
```

This method may suit your book search better because books are normally sorted alphabetically! If you know the alphabet and the positional number of each letter you could easily find any author you wanted!

Min and Max algorithms

How do you find the biggest and smallest numbers in an array?

Well, first: why is this useful? Can't you just look at a list and tell? Yes, *you* can but a computer cannot.

Finding maximum from n +ve (+ve means positive) numbers

So given an array, A, of positive only numbers this is how you would find the maximum.

Quick note: read ":" as "then".

```
i = 1
m = 0
while i <= n:
    if A[i] > m:
        m = A[i]
    i = i + 1
print(M)
```

Assume A is an array of n positive numbers and m is the maximum number. Also note that arrays are indexed at 1 here. Normally in computer science arrays are indexed at 0, as in they start from 0.

Can you guess the time complexity of this?

It's O(N).

Let's try a quick example. Given the array [2, 6, 3, 4] how does the computer know what number is biggest? How best can we break down this problem into little chunks?

Well, we'll simply run the code.

At first, 2 is the largest number. Then 6 is, 3 isn't larger than 6 and 4 isn't larger than 6 so we return 6 as the largest number in the array.

Finding minimum from N +ve (positive) numbers

What if we wanted to find the minimum? Well, we simply turn the ">" sign into a "<" sign.

```
i = 1
m = A[1]
while i <= n:
    if A[i] < m:
        m = A[i]
    i = i + 1
print(m)
```

Findig the maximum number from N +ve (positive) numbers

```
i = 1
m = A[1]
while i <= n:
    if A[i] > m:
        m = A[i]
    i = i + 1
print(m)
```

You literally just change "if A[i] < m" to "if A[i] > m"

Dipping our toes into data structures - Queues and Stacks

A datastructure is a way to structure data in such a way that the data becomes easily usable and maybe even faster to use than a typical list or array.

Queues

A queue is a first-in-first-out array. The *first* item into the array is the *first* item out of the array.

Queues have 2 functions:

- Enqueue - Insert element to tail (end).
- Dequeue - Retrive data from head (start) of queue.

So given an array (list) as an example

```
[15, 20, 10, 0]
```

The *head* points to array[1] which is the first item in the array, it does not point to the data contained in the first item. The *tail* points to array[end], where end is the length of the array + 1. In this instance tail results in 5,for their are 4 items in the list and then you add 1.

It's actually up to the developer who designs this as to whether tail points to length + 1 or whether it points to length.

The reason tail always points to something that doesn't exist is because enqueue is implemented to always put an object at [tail]. If we were to make tail the same as the length than enqueue would be changed to [tail + 1]. Either way you are adding 1, so it does not matter and may change in some languages.

If we want to *enqueue* 12 into the queue, we would add it onto the end like so:

```
[15, 20, 10, 0, 12]
```

And the tail increases to a length of 6, but head stays at 1.

Dequeue takes the [head] of the list and enqueue places an item at the [tail] of the list.

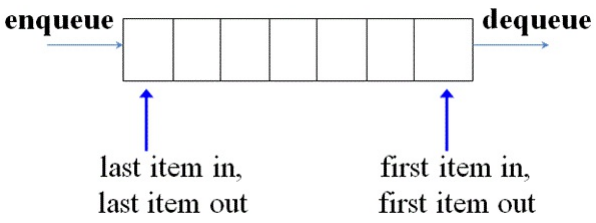
```
x = [20, 10, 0, 12]
```

which results in head = 2 and tail = 5. The head variable increased because we dequeued something. The head variable is also useful for other things such as knowing how many items have been taken out of a queue.

If the head and the tail are the same and you have a 1 dimensional array like so:

```
x = [1]
```

then in some programming languages you can only get the head() of the queue (Haskell, i'm looking at you) but in other languages this may differ.



Why are queues useful? Well, queues are extremely useful. Imagine lining up at the bank and waiting 30 - 40 minutes to get to the front. When you get to the front, the teller decides a queue isn't useful so they start from the back, making you the last person they reach.

Stacks

A stack is a *last-in-first-out* array.

In a lot of online tutorials stacks are drawn vertically like so:

0 :---: 1 2 3

Much like a stack of plates, the first item on the top is the first item to come off. You can't pull plates out of the bottom of a stack of plates (if you can, you should become a magician!).

Stacks have 2 functions:

- Push - Insert element to the location top + 1
- Pop - Delete element from top

Where "top" is the variable for the top-most item in the stack.

Let's see some examples.

```
x = [20, 10, 15]
```

Where top is "3" because there are 3 elements.

So to push 12 onto this stack we'll get

```
x = [20, 10, 15, 12]
```

Now top becomes 4.

You may be wondering "what happened to adding +1 to head?". Well, like I said, you can use both notations. I want to expose you to as much variance in algorithms as possible to enrich your learning... Well, at least that's what my professor said to me.

Now if we want to pop() a stack we'll get:

```
x = [20, 10, 15]
```

and head = 3, since we've popped the top!

Stacks are super useful, especially in browser history. Say for example you go to Google, then Medium, then my profile (follow me 😉). The stack will look like:

Brandon's Profile :----------: Medium Google

Now how much would it suck if you pressed "back" and it went back to Google? It would suck alot! So when we click "back" the browser, quite literally, pops the current webpage off of the stack and brings you to the next item.

Stacks grow *downwards* by placing stuff on top of it. This may sound confusing, so I'll try to explain it.

When you *push* something onto a stack, you're changing the head of the stack (the head is normally the first item) to go down 1 level. We'll visualise this. Let's imagine a stack that looks like this:

```
[a][b][c]
```

There are no rules as to whether stacks can be sideways or top down, but normally they're top down. If we want to *push* something into the stack we do this:

```
[x][a][b][c]
```

Which pushes all the contents of the stack down.

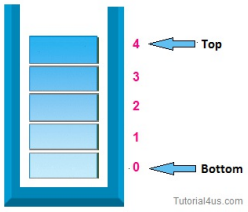
In normal stack-like behavior this looks like:

Follow me on Twitter :----------: Medium Google

Now we push "University of Liverpool" to the stack and it becomes:

University of Liverpool :----------: Follow me on Twitter Medium Google

Literally pushing the stack down.



Think of stacks like leaving breadcrumbs for yourself. If you're ever lost in a maze and you place down all the breadcrumbs, you'll look for the last breadcrumb you placed down, which is usually the one closet to you.

Linked Lists

Linked Lists are a linear collection of data elements except the linear order is not defined by their physical placement in memory but instead each data node points to the next.

Linked lists are much faster for inserting data into because you don't need to shift the entire array to add an item. They're very useful for when the space of the plausiable values could be very large but you don't know when you will grow to that size.

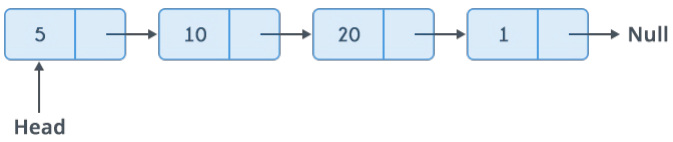
Linked lists are scalable and adaptable.

The order is determined by a pointer (rather than array indices) TK Each element (node) has a data field and one or two pointers linking to the next or previous elements in the list

There are two types of linked lists, singly linked and doubly linked. Singly link looks like

```
[15][-]->
```

Where `->` represents a pointer and each `[]` represents a componenet.



In a singly linked list the node stores one piece of data (the 15) and it stores a pointer linking it to the right hand side. The pointer does not contain data, just a pointer. In my badly drawn diagram the pointer, the arrow which is literally pointing out of the right hand side box is a pointer pointing to the next node. Each set of these data with pointers is a node.

A singly linked list cannot point to the previous node. Doubly linked list looks like:

```
<-[-][15][-]->
```

In a doubly linked list the node has a forward and backward pointer as well as a data object. Each node has 3 componenets, 1 piece of data and 2 pointers. In a doubly linked list you can go to the previous node.

Linked lists have a few functions we can use such as:

- node.data = get data from current node

- node.next = go to next node
- node.prev = go to previous node

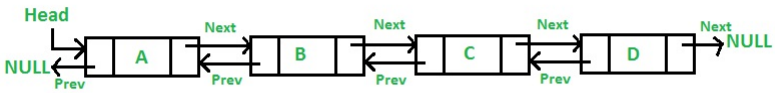
The notation used may be different in programming languages.

Note: if the node you are looking at is the last node in the linked list then node.next results in None (as in, it does not exist) and if you try to run node.prev in a singly linked list it will error.

```
head -> [ ][15][<-]---[->][10][<-]---[->][20][ ]  <- tail
```

Something important to stress again is that a node in a doubly linked list has *3 components* and each [ ] represents 1 component so a group of [ ][ ][ ] represents *1 node* and in this diagram there are *3* nodes respectively.

If my badly-drawn diagrams in ASCII art are confusing then this picture may help:



Traversing Linked Lists

We can traverse and output each element of a linked list like so:

```
node = head
while node != None: # while node does not equal None, where != is not equal to
    print(node.data) # output the data of the node using the Python print function
    node = node.next # updates the node variable to contain the next node in the list
```

Notice how we are using the node.next function to traverse a linked list.

At Merriam Webster they have dictionaries with reversed words. So Ecology would be ygolcoe. If you wanted to know how many words ended in "cology" you simply would search all words that ended in "ygolco". Of course, reading it like this is annoying so you can use a doubly linked list to be able to search a dictionary both the original alphabetical way and the reverse way. Thanks 99PI.

You could probably program a data structure (doubly linked list) that when you go to a previous node it displays the data in reverse.

Big O

Searching for and traversing through linked lists is O(n).

Programming Linked Lists

Because the linked lists data structure isn't in every language (very commonly it is not) we have to program linked lists ourselves.

Unfortunately my lectuer decided to switch to Java at this point in the class, however I'll do my best to explain every piece of java code.

So below is a node for a doubly linked list. The node has 3 components, previous, data, next. A class is a template you can apply to objects, so in this instance we can make new nodes using this class.

```
class Node {
        public int data;
        public Node next;
        public Node prev;

        // constructor to create a new node with data equals to parameter i
        public Node (int i) {
                next = null;
                prev = null;
                data = i;
        }
}
```

In this instance the node has 3 methods that can be applied to it. Each method can be applied like:

```
Node.next
Node.prev
Node.data
```

using the dot notation.

If we wanted to make a single node in a doubly linked list we can do this:

```
Node newnode = new Node(5);
```

Where the node will hold a data value of 5. If we wanted to add a new node to the front of the list we can use this function:

```
// create a new node containing data value
// and insert the new node to head of the list
static void insertHead(int value) {
        Node newnode = new Node(value); // makes a new node with value

        newnode.next = head; // Makes the next node the head of the linked list
        newnode.prev = null; // makes the previous node empty
        if (head != null) // if the head is empty
                head.prev = newnode; // sets the previous node to the new node to be inserted
        else
                tail = newnode; // the last node in the list is now the new node
        head = newnode; // the head of the list is the new node
        }
```

This code will create a new node and insert this node at the front of the doubly linked list. This is quite confusing, so let's draw some diagrams.

First we create a new empty node with value as data.

```
[ ][value][ ]
```

Then we set the next pointer to point at the head of the linked lists

```
[ ][value][-]-[>][HEAD][ ]
```

After this we set the previous definition to be NULL

```
NULL<-[-][value][-]-[>][HEAD][ ]
```

Then we run a simple if statement. If head is not empty, set the previous node of the head node to be our new node.

```
NULL<-[-][value][<-]-[>][HEAD][ ]
```

Notice how you can now travel from the HEAD node to the new node.

If head is empty, set the tail (last) node to be the new node. This is because if you insert this node into a linked list that doesn't exist, the next node would be nothing so the node you just inserted will be both the head and tail. Otherwise, just ignore the next node as it contains data.

The last line sets the head pointer to point to the new node which is at the front of the list. We can also delete a node at the front of the linked list in a similar fashion:

```
// delete the node at the head of the linked list
static Node deleteHead() {
        Node curr;

        curr = head;
        if (curr != null) {
                head = head.next;
                head.prev = null;
        }
        return curr;
}
```

We assume here that curr is a pointer that points at a node in the linked list.

So we point the current pointer at head, if head is not null (if it is not empty) then we set the head to the next node and from this new head (which is the second item in the linked list) we set the previous (old head) to null (nothing, doesn't exist). We then return the pointer.

Inserting items into a linkedlist

Linked lists are super cool because we can insert items anywhere in them. Let's say we have a pointer, curr, which points somewhere (let's say the middle) of the list. We can insert a new node after curr like so (in Python now)

```
def listInsert(L, curr, newnode):
    newnode.next = curr.next
    newnode.prev = curr
    if curr.next != None:
        curr.next.prev = newnode
    curr.next = newnode
```

Whenever we want to insert a new node, we just have to tell the node what the next and previous nodes are.

Searching over a sorted linked list

Recall that values stored in a linked list are sorted. We could use binary search to search the list. However, this is a bad idea. We don't know where the middle of a linked list is. Everytime we wanted to find the middle we would have to count every single node in the list and half that by 2.`

We can use a modified version of sequential search to search a linked list.

Because the linked list is sorted, let's say in ascending order, we can use this information to make seqential search faster.

```
node = head
while node != None and node.data < key:
    node = node.next
if node == None:
    print("Not found")
else if node.data > key:
    print("not found")
else:
    print("found")
```

Since the linked list is sorted sequentially we know that the nodes in the linked list go in some order, like 1, 2, 3, 4, 5 for example. If node.data is more than the key (what we're looking for) we know it's not in the list, because it is sorted in some order.

There are many, many search algorithms but most of the time if you know a little bit of information about the data you can change some search algorithm to be more efficient for that specific problem. In general, binary search is extremely effective but here it's not so good. Don't just use an algorithm because Stack Overflow says that it is the fastest, best algorithm for the job.

Algorithms are like programming languages, we all have our favourites and sometimes we say that one programming language is better than another (Python, I love you) but at the end of the day it would be foolish and naieve to say that one programming language is better than all the others. Use the right tool for the job, and change it if you want to!

Personally whenever I am presented with a problem I like to imagine it is in its own seperate world. I try to imagine what rules of life apply to this world. Once you understand the rules

Sorting Algorithms

Let's say you have a bookshelf and you want to arrange the books alphabetically, this is a sorting problem and the way you go around sorting the books is a sorting algorithm.

Bubble Sort

The idea of a bubble sort is simple. Starting from the first element, swap adjacent items if they are not in ascending order. When the last item is reached, the last item is the largest. Repeat these steps for the remaining items to find the second largest, third largest and so on.

Bubble sort is often the first sorting algorithm one learns because it's easy to implement.

Here's a new gif of how Bubble Sort works.

Big O

The Big O complexity for bubble sort is $$O(n^2)$$.

Code

Here is some Python code for the bubble sort:

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
```

And here is the Java version:

```
public static void BubbleSort( int [ ] num )
{
     int j;
     boolean flag = true;   // set flag to true to begin first pass
     int temp;   //holding variable

     while ( flag )
     {
            flag= false;    //set flag to false awaiting a possible swap
            for( j=0;  j < num.length -1;  j++ )
            {
                   if ( num[ j ] < num[j+1] )   // change to > for ascending sort
                   {
                           temp = num[ j ];                //swap elements
                           num[ j ] = num[ j+1 ];
                           num[ j+1 ] = temp;
                          flag = true;              //shows a swap occurred
                    }
            }
      }
}
```

You'll notice that bubblesort heavily uses a temporary variable for temporary storage.

Selection Sort Algorithm

The selection sort algorithms tries to do the following:

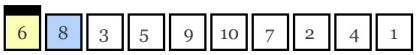
- Find minimum key from the input sequence
- Delete it from input sequence
- Append it ot the resulting sequence
- Repeat until nothing left in input sequence

Example, where bolded means it's in the right place.

[34, 10, 64, 51, 32, 21]	**To Swap**
[34, 10, 64, 51, 32, 21]	34, 10
[*10*, 34, 64, 51, 32, 21	34, 21
[*10*, *21*, 64, 51, 32, 34]	32, 64
[*10*, *21*, *32*, 51, 64, 34]	34, 51
[*10*, *21*, *32*, *34*, 64, 51]	51, 64
[*10*, *21*, *32*, *34*, 51, 64]	Fully sorted

As you can see there is fewer swaps than in Bubble Sort. It basically finds the lowest value and swaps it with the closet to first in the list where it isn't in the right place.

Here's a gif representing how selection sort works:



Yellow is smallest number found
Blue is current item
Green is sorted list

Big O

The Big O notation for Selection Sort is O(^2)

Code

Here's some Python code for selection sort:

```
def selectionSort(alist):
   for fillslot in range(len(alist)-1,0,-1):
       positionOfMax=0
       for location in range(1,fillslot+1):
           if alist[location]>alist[positionOfMax]:
               positionOfMax = location

       temp = alist[fillslot]
       alist[fillslot] = alist[positionOfMax]
       alist[positionOfMax] = temp
```

Insertion Sort

The idea is as follows:

- Look at elements one by one
- Build up sorted list by inserting the element at the correct location

Hence why it's called insertion sort, because you are inserting hte values into a new array.

[34, 10, 64, 51, 32, 21]	**No. shifted to right**
[34, 10, 64, 51, 32, 21]	Nothing yet
[10, 34, 64, 51, 32, 21]	34
[*10, 34*, 64, 51, 32, 21]	Nothing yet
[*10, 34, 51*, 64, 32, 21]	64
[*10, 32, 34, 51, 64*, 21]	34, 51, 64
[*10, 21, 32, 34, 51, 64*]	32, 34, 51, 64

Here's a gif showing how Insertion Sort works

Big O notation

The Big O notation for Insertion Sort is O(n^2).

Code

```
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position > 0 and alist[position - 1] > currentvalue:
         alist[position] = alist[position - 1]
         position = position - 1

     alist[position] = currentvalue
```

Programming - Linked Lists

Bubble Sort with Linked List

Given the linked list

```
cur
[ ][34][-]---[>][10][-]---[>][64][-]---[>][21][ ]
```

The word "cur" is a pointer pointing at the current item being looked at.

We compare 34 and 10 and becaue 10 is smaller than 34, we swap them.

```
           cur
[ ][10][-]---[>][34][-]---[>][64][-]---[>][21][ ]
```

And then we just continue this until we finish it. 34 < 64, so no swaps.

```
                        cur
[ ][10][-]---[>][34][-]---[>][64][-]---[>][21][ ]
```

```
                                    cur
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

Cur moves through the linked list by using the ".next" function we saw earlier. But in this instance, Cur cannot move back because Cur.next is NIL, it is nothing. You cannot go any further. You also can't go back, because this is a singly linked list. So what do we do?

We want to move Cur back to the start but we also don't want to look at the whole list, just the second last one because the start of the list has already been sorted.

In the first round, we want to see if we reach the last possible element. In the second round we want to know we are at the second to last item. But how do we know this?

We need to create a "last" pointer to point to where we say "last" is.

```
                                    cur       LAST
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

In the next round we change the last variable to point to the last node, because we know that's where LAST is.

```
Curr                                  LAST
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

```
              cur                     LAST
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

```
                        cur           LAST
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

Here we swap 34 and 21 over.

```
                        cur           LAST
[ ][10][-]---[>][21][-]---[>][34][-]---[>][64][ ]
```

We have to make an expression, cur.next == last. If this equates to true, than we know we're at the last (what we say is last) part of the list. We then move last to cur and go back over the list:

```
cur                     LAST
[ ][10][-]---[>][21][-]---[>][34][-]---[>][64][ ]
```

Then we simply go through the linked list again

```
               cur          LAST
[ ][10][-]---[>][34][-]---[>][21][-]---[>][64][ ]
```

Now cur.next == last, so we stop. We've sorted the whole list.

Programming a bubble sort for a linkedlist

Here is some psuedocode for a bubble sort in a linkedlist:

```
if head == NIL then:
    empty list and STOP!
last = NIL.curr = head

while curr.next != last:
    while curr.next != last:
        if curr.data > curr.next.data:
            swapNode(curr, curr.next)
        curr = curr.next

    last = curr
    curr = head
```

Time complexity is O(n).

Here's the Java code for the bubble sort. This is the code for a single node in a linked list:

```
class Node {
        public int data;
        public Node next;
        public Node prev;

        // constructor to create a new node with data equals to parameter i
        public Node (int i) {
                next = null;
                prev = null;
                data = i;
        }
}
```

And this is the code for a bubble sort on a linked list:

```
import java.util.*;
import java.io.*;


// Implement a linked list from the object Node
class SampleLinkedListBubbleSort {

        public static Node head, tail; // head and tail of the linked list

        public static void main(String[] args) throws Exception {
                Node curr;

                // some arbitrary input to test the program

                insertHead(15);
                insertHead(20);
                insertHead(10);
                insertHead(25);
                insertHead(30);
                printList();


                bubbleSort();

                printList();

        }

        // bubble sort
        static void bubbleSort() {
                Node last, curr;

                System.out.println("Bubble Sort ...");
                if (head != null) {
                        last = null;
                        curr = head;
                        while (curr.next != last) {
                                while (curr.next != last) {
                                        if (curr.data > curr.next.data)
                                                swapNode(curr, curr.next);
                                        curr = curr.next;
                                }
                                last = curr;
                                curr = head;
                        }
                }
        }


        static void swapNode(Node a, Node b) {
                int tmp;
                tmp = a.data;
                a.data = b.data;
                b.data = tmp;
        }

        // create a new node containing data value
        // and insert the new node to head of the list
        static void insertHead(int value) {
                Node newnode = new Node(value);

                newnode.next = head;
                newnode.prev = null;
                if (head != null)
                        head.prev = newnode;
                else
                        tail = newnode;
                head = newnode;
        }

        // delete the node at the head of the linked list
        static Node deleteHead() {
                Node curr;

                curr = head;
                if (curr != null) {
                        head = head.next;
                        head.prev = null;
                }
                return curr;
        }

        // print the content of the list in two orders:
```

```
        // (i) from head to tail
        // (ii) from tail to head
        static void printList() {
                Node curr;

                curr = head;
                System.out.print("From head: ");
                while (curr != null) {
                        System.out.print(curr.data + " ");
                        curr = curr.next;
                }
                System.out.println();


                curr = tail;
                System.out.print("From tail: ");
                while (curr != null) {
                        System.out.print(curr.data + " ");
                        curr = curr.prev;
                }
                System.out.println();

        }

}
```

Selection Sort with Linked List

TK add code

So it's the same example with bubblesort but we want to see it done with a selection sort:

```
[ ][34][-]---[>][10][-]---[>][64][-]---[>][64][-]---[>][21][ ]
```

The "min" pointer loops through the entire linked list to find the *smallest unsorted* node in the linked list

```
curr            min
[ ][34][-]---[>][10][-]---[>][64][-]---[>][64][-]---[>][21][ ]
```

The nodes don't entirely swap over here, you just swap the data of each node over using a temporary value. The idea is that you always placed the *smallest unsorted* node (data of a node) in the correct place until the "curr" pointer points at the end of the list (we know it points at the end when curr.next equates to NIL, None, Null).

Programming a Selection Sort

```
// selection sort ascendingly
// on array[0] to array[n-1]
// n is length of array
static void select_sort(int[] array, int n) {
    int minloc;
    int num_comp, num_swap;

    num_comp = 0;
    num_swap = 0;
    System.out.println("Selection sort...");

    // selection sort on array[], using swap()
    int i = 0;
    int loc = 0;
    int j = 0;

    while (i <= n - 1){
        loc = i;
        j = i + 1;

        while (j <= n - 1){
            num_comp++;
            if (array[j] < array[loc]){
                loc = j;
            }
            j = j + 1;
        }
        if (i != loc){
            swap(array, i, loc);
            num_swap++;
        }
        i = i + 1;
    }



    print_array(array, n);
    System.out.println("Number of comparisons is " + num_comp);
    System.out.println("Number of swaps is " + num_swap);
}
```

Insertion Sort with Linked List

So given a vector <34, 10, 64, 21> we want to insert these numbers into the right order. We start with 34 so we have:

```
head
[ ][34][ ]
```

Now we have 10, which is the new smallest element so it gets inserted to the start of the linked list like so:

```
head
[ ][10][-]---[>][34][ ]
```

64 is the largest number we've seen so we insert it at the very end. We know it's the largest number becaues we can either iterate over the list to check or keep a MIN_VALUE variable which tells us what the smallest number is we have inserted into the linked list.

```
head
[ ][10][-]---[>][34][-]---[>][64][ ]
```

The next round we need to find a node that is smaller than our target (21) and larger than our target (21). So we insert it into the middle after finding this:

```
head
[ ][10][-]---[>][21][-]---[>][34][-]---[>][64][ ]
```

Okay, what if we get given some new data and we want to sort this appropriately?

We need to assume that the linked list is already sorted ascendantly, and we want to insert a node in the proper position.

If the list is empty OR the first element (head.data) is larger than what we want to insert (node.data) we should put the new node as the head of the list.

Else we should check element by element to find one larger than node.data.

Programming an Insertion Sort

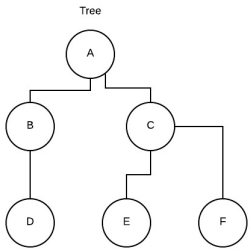
Here's some Java code for this.

```java
// selection sort ascendingly
// on array[0] to array[n-1]
static void insert_sort(int[] array, int n) {
    int loc, key;
    int num_comp, num_shift;

    num_comp = 0;
    num_shift = 0;
    System.out.println("Insertion sort...");

    // insertion sort on array[], using swap()

    for (int i = 1; i <= n - 1; i++){
        num_comp++;
        key = array[i];

        loc = 0;


        while (loc < i && array[loc] < key){
            loc = loc + 1;
        }
        for (int j = i; j >= loc+1; j--){
            num_comp++;
            array[j] = array[j - 1];
        }=

        array[loc] = key;
    }

            print_array(array, n);
        }
```

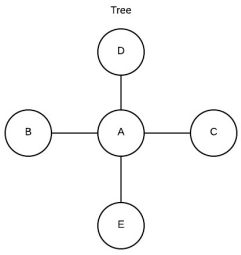
Tree Data Structure

A tree T = (V, E) consists of a set of vertices (V) and a set of edges (E).

The vertices are the "Nodes" in a tree and the edges are the "lines connecting them".

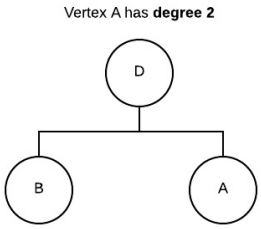


And this is also a tree:



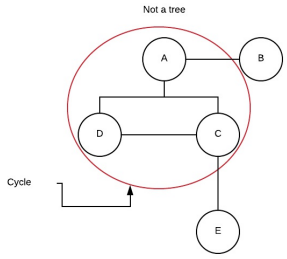
Tree's have **exactly** one path between two vertices. You cannot have more than one path between any 2 vertices.

The number of paths that lead into and out of a vertice is called the **degree** of a vertice.

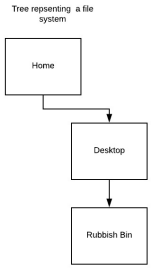


A **cycle** is where you can "Move all the way around". Notice the vertices y, u, and v. You can "move all the way around" these 3 vertices. **Acyclic** means there is no cycle in the graph. If it is Acylic it implies it is a tree.

If it has more than 2 paths from 1 vertex to another or it has a cycle then it is not a tree.



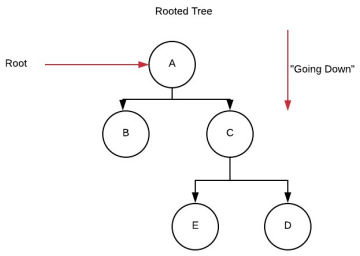
A tree is often used to represent something that has a hierarchical sturcture, such as files and folders in a desktop.



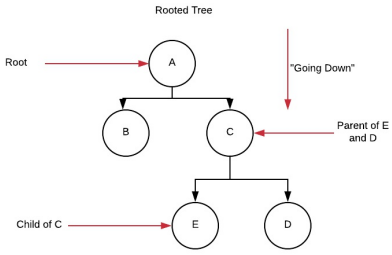
A **rooted tree** has a direction where it goes from the top to the bottom but in some cases we can have an **unrooted** tree where it is not drawn top to bottom.

TK image here of unrooted tree

The topmost vertex is called the **root** of the tree. Where it all comes from.



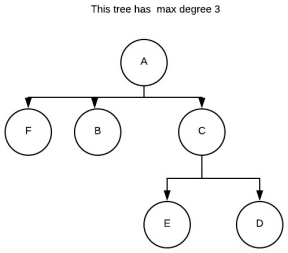
A vertex might have **children** below it, connecting to the vertex. In this case we say the ones below it are the **children / child** and the original vertex is the **parent** of the vertex.



In a **rooted tree** there is an implicit direction of the tree that is often not drawn in rooted trees, usually it goes downwards.

The **degree of a vertex** is the number of children it has.

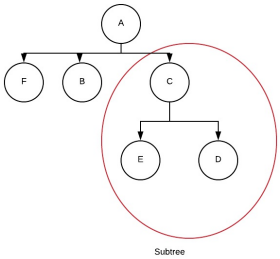
The **degree of a tree** is the max degree from a vertex in the tree. So if a vertex has a degree of 3 and no other vertex has a degree higher than 3 then the degree of the tree is 3.



A vertex with no children (degree 0) is called a **leaf**.

Vetrices other than leaves / roots are called **internal vertices**. These are sometimes called downward branches in a **rooted tree**.

A **subtree** is a tree that comes from a vertex that isn't the **root** vertex. You can have a subtree of a subtree.

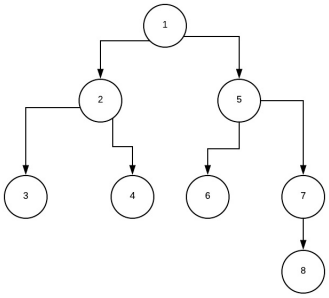


Any vertex can be considered a sub-tree with 1 single leaf in it.

Binary Tree

A binary tree has a degree of most 2. No vertex has a degree higher than 2. The two subtrees are called the left subtree and the right subtree.

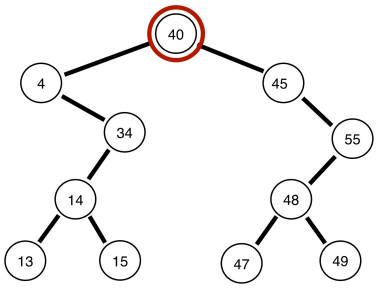
The left hand side tree is smaller, the right hand side tree is larger.



You can traverse binary trees using these 3 algorithms, but note that these only work on binary trees because we need to have a "left" subtree and a "right" subtree.

Pre-order Traversal

A preorder traversal method visits the left subtree first, then the right and then eventually the right subtree. It tries to go down as far left as possible and stick to the left hand side as much as possible.



Gif from here.

In-order Traversal

The left subtree is visited first, then the root vertext and then the right sub-tree is visited.

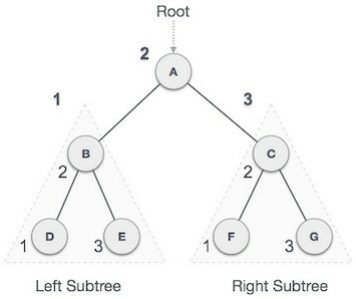


Image from here

In-order traversal gives us numbers in ascending order.

Post-order Traversal

We first traverse the left subtree, and then the right subtree, and then finally the root node. The root node is visited **last**.

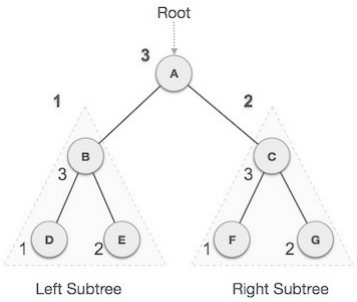
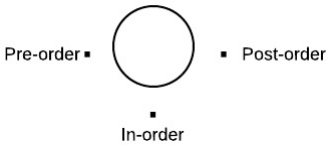


Image from here

https://www.youtube.com/watch?v=GJ9rSCZsTEw TK watch this

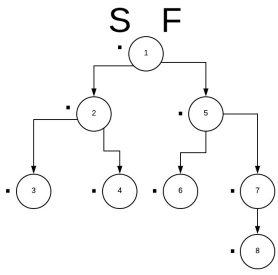
What if you forget which one's which?

Luckily there's this "dot" notation you can use!



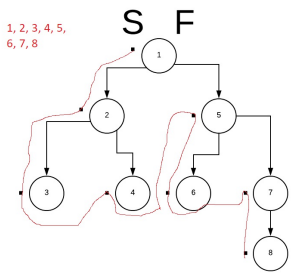
So, firstly you need to know these dot positions. They'll come in handy in a second!

So let's say you want to find the the pre-order traversal, you put the dot to the left of the node like in the picture above.



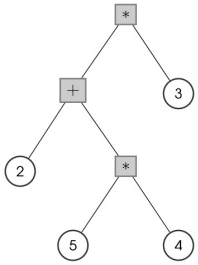
Then you write S on the left hand side of the root node and F on the right hand side of the root node.

Now you simply draw a line from S to F



Try not to go back on yourself. We could of gone to 6 and then 5, but that would of made a mess and it would be hard for us to go back on ourselves without the lines looklikg like they touch.

Expression Trees



A mathematical expression can be expressed as a tree like so.

In fix notation is written as:

$$ (2 + 5 * 4) * 3$$

This is what we normally use. The operator (addition, multiplication) is **in** the expression.

Postfix notation looks like:

$$ 2\space 5\space4 * + 3 * $$

We can use post-fix traversal to get the postfix notational representation of

The operator is at the end of the expression it is evaluating. Computers often use this notation more than in-fix notation.

If we look at the in order traversal for the expression tree we get the in fix notation. If we want to get the post fix we need to use post order traversal.

Eviction Algorithms

Let's say you have a slow hard drive, k, which contains the following data:

$$ k = [4, 5, 3, 7, 9, 4] $$

And you have a faster memory storage, called a **cache**, n, that has the rule n < k and n is a subset of k. So in other words, the cache can never be the same size as the hard drive or larger than the hard drive and all items in the cache must be in the hard drive. So we can have a cache like this:

$$ n = [4, 5, 3, 7] $$

And then in this instance we can say that cache can only hold 4 items.

What if we wanted to add another item to the cache? What if we use the number "5" a lot?

We would want to **evict** something from the cache in order to allow us to put another item into the cache.

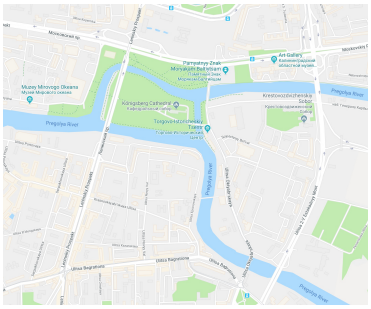
No Eviction

The easiest algorithm would be to not evict any data at all. This is bad because that means that the data in the cache will stay that way forever. But obviously humans over time evole, what if we **never** use the number "4" but we always

Graph and Graph Theory

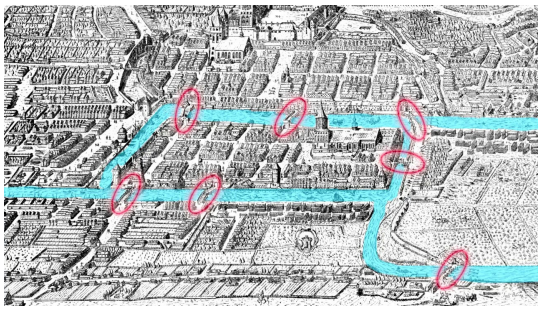
If you want to learn **a lot** about Graph Theory, check out this article

The seven bridges of Konigsberg is the foundation and birth of graph theory.



There was a puzzle that stated:

> Can you cross all seven bridges exactly once?



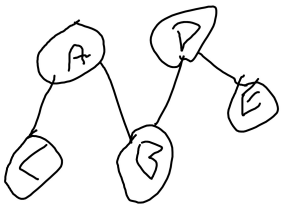
There are 2 rules for this problem:

1. Do not cross any bridge twice
2. All bridges must be crossed

In the 18th Century a mathematician called Euler realised this problem was impossible. Every bit of land you enter has to have 2 bridges, or an even number of bridges. One you can leave on, one you can enter on.

You'll notice a part of the land does not have an even number of bridges, it actually has 3 bridges.

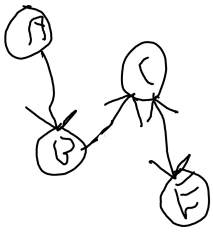
An **undirected** graph G = (V, E) consists of a set of vertices V and a set of edges. It is an undirected graph because the edges do not have any direction.



Excuse my messy drawing but it is incredibly hard to draw on Paint with a touchpad.

Each edge is an unordered pair of vertices. So {a, b} is the same as {b, a}.

A **directed** graph G = (V, E) is where each vertice has a direction.



Think of it like Facebook and Twitter. On Facebook when you friend someone, the other person is automatically a friend of you.

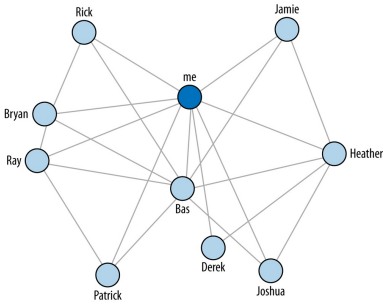


image from here

Graphs are used to model computer networks, state spaces of finite games such as Chess.

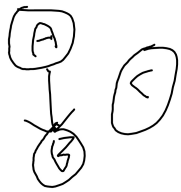
Here are some of the different types of graphs:

Simple Graph

The Simple Graph has at most 1 edge between 2 vertices and it has no self-loop. It has no edges that come from a vertex and go back to that same vertex.

This is **not** a simple graph:

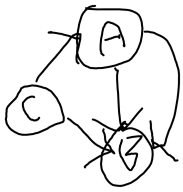
And this is not a simple graph, because a vertex exists with no edges connecting to it:



This is a simple graph:

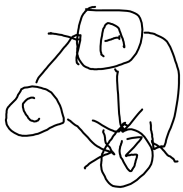
Multi Graph

A multi graph allows more than one edge between two vertices:



More on Undirected Graphs and Terminology

In an undirected graph, G, suppose that e = {u, v} is an edge of G



u and v are said to be **adjacent** and are called **neighbours** of each other

e is said to be **incident** with u and v

u and v are called **endpoints** of e

e is said to **connect** u and v

The degree of a vertex is how many edges are connected to it.

The degree of the graph is the maximum edges connected to a particualr vertex. In this graph the degree is 3, since vertex u has degree 3 and is the largest degree in the graph.

Matrix Representation of Graphs

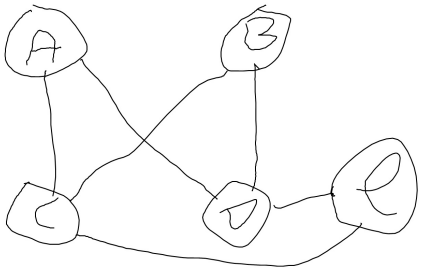
An undirected graph can be represented by an adjanecy matrix.

A matrix is like a vector or a set, it's a storage unit to store numbers in it.

An **adjacency matrix**, M, for a simple undirected graph with n vertices is called an **n x n matrix**.

In this matrix if vertex i and vertex j are adjacent (neighbours) then you can represent this on the matrix with the number 1.

If they are not, use the number 0.



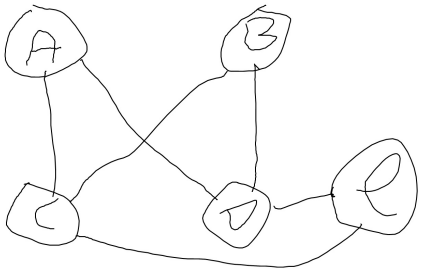
To represent this in a matrix, we can do the following:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

Notice how the diagonal is 0's and if you take half of the upper triangle it matches the bottom half.

An **incident matrix** is an m x n matrix where m is the number of edges in the graph.

For this graph again:



We can use this incidence matrix to represent it:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

More on Directed Graphs

An **in-degree** of a vertex, v, is the number of edges leading to v.

An **out-degree** of a vertex, v, is the number of edges leading away from v.

The **in-degree** is the same as the **out-degree**. It's also the same as the number of edges.

$$ in-degree-sum = out-degree-sum = number-of-edges$$

A directed graph can be represented by an adjacency matrix or an incidence matrix.

Adjacency Matrix

An **adjacency matrix**, M, for a directed graph with n vertices is called an n x n matrix.

- M(i, j) is equal to 1 if (i, j) has an edge from i to j
- M(i, j) is otherwise 0.

An **adjacency list** is where each vertex, u, has a list of vertices pointed to by an edge leading away from u.

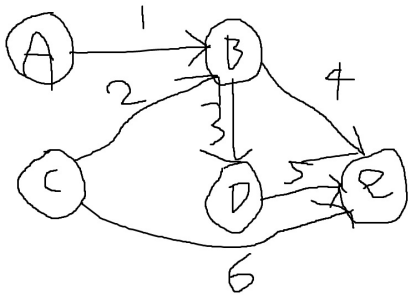
This is really nothing different from what we saw earlier.

Incidence Matrix

An **indidence matrix** for a directed graph with n vertices and m edges is an m x n matrix.

These are the basic rules:

- M(i, j) = 1 if edge i is leading away from vertex j (leaving)
- M(i, j) = -1 if edge i is leading to vertex j (into)
- M(i, j) = 0 otherwise



	a	b	c	d	e
1	1	-1	0	0	0
2	0	-1	1	0	0
3	0	1	0	-1	0
4	0	1	0	0	-1
5	0	0	0	1	-1
6	0	0	1	0	-1

Incidence list is a list wheere each vertex, u, has a list of vertices pointed to by an edge leading away from u.

Circuits

A circuit, a path, a cycle are all sequences of vertices and edges.

They all have rules and properties which make them special, these are:

- Cycle: Vertices cannot repeat. Edges cannot repeat.
- Walk: Vertices may repeat. Edges may repeat.
- Circuit: Vertices may repeat. Edges cannot repeat.

Normally a circuit is defined as a path from vertex a, back to vertex a.

A **simple** circuit visits an edge at most once (so never goes back to the same vertex).

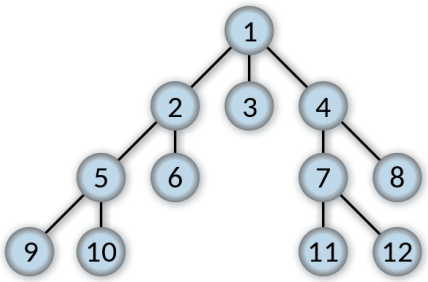
An **Euler circuit** is a circuit visiting every edge exactly once (so can go back to the same vertex).

Searching on Trees / Graphs

Okay, so we've met trees and graphs. But how do we search them? We can use some of these nifty search algorithms!

Breadth First Search

Breadth First Search (BFS) is a search algorithm developed by Konrad Zeus for his rejected PhD thesis in 1945. Breadth First Search searches all neighbors before it searches child nodes. In the below picture, once the start state (1) has been searched the states 2, 3, and 4 will then be searched.



The Breadth First Search Algorithm has a queue which is vital to how it works. Breadth first will first check whether the current node it is searching is the goal state or not. If it is not the goal state, it places all child nodes of the current node being searched into a queue. As an example assume the queue will look like [5, 6, 3, 4].

Because of the first in first out nature, the first ones added to the queue are the first ones out of the queue, so it would search in the order 4, 3, 6, 5.

Breadth first search searches in "levels". It starts at level 1, [1], then goes down to level 2, [1:2, 1:3, 1:4].

Advantages

Breadth-first search is complete, as in it will always find a path and the shortest path to the goal, assuming the goal is at a finite depth.

Space and Time Complexity

Time complexity is how long it takes the algorithm to run given an input, usually denoted in Big O notation. Space complexity is how much the algorithm takes up in memory. Although this depends on the hardware factors, just like with Big O notation we can use a notation to represent how much space it'll take up.

Consider a theoretical tree where node state has b successors. The root of the search tree generates b nodes and the second level of the search tree generates b^2 nodes. Each level generates b more nodes, yielding b^n (b to the power of n) nodes where n is the level the search tree is on. So the **time complexity** is $$b^n$$.

The **space complexity** of this is $$b^d$$.

You may notice this looks different from Big O notation, well, for some reason a lot of AI researchers use this notation. In big O the space and time complexity is:

$$ O(|v|)$$

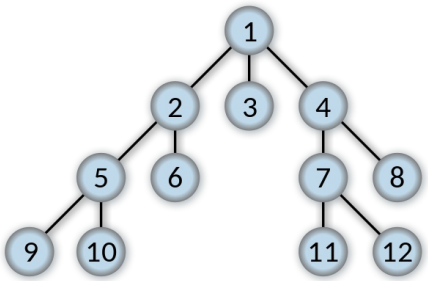
Where |v| is the number of nodes.

Disadvantages

Breadth First Search is very, very slow and requires a lot of memory.

Depth First Search

Depth First Search expands the deepest node in the current frontier first. Depth first search goes immedtially to the deepest possible point of the search tree until there are no sucessors.



In this example, Depth First Search will go straight to 9, then 10 and then to 6.

Whereas Breadth First search uses a first in first out (FIFO) queue Depth First uses a a Last in Last out queue (LIFO). A LIFI queue means that the most recently generated node is chosen for expansion. The most recently generated node must be the deepest possible unexpanded node because it is deeper than its parent node.

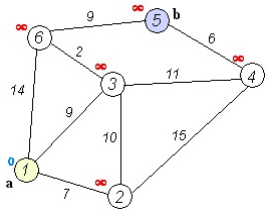
If Depth First Search is used on a graph which avoids repeated states and redundant paths then it will find its goal in a finite number of states. If however it is used on a search tree then it will expand forever, in other words depth first search is not complete within search trees.

Depth first search will not find the optimal path.

The time complexity of DFS is 1+ b^2 + b^3 + ... +b^m

The advantage of DFS over BFS is the space complexity. Once a path has been fully explored it can be removed from memory, so DFS only needs to store the root node, all the children of the root node and where it currently is. DFS requires space complexity of bm where b is the branching factor and m is the longest path in the graph.

Uniform Cost Search



Uniform Cost search is Dijkstra's Algorithm but rather than finding the single shortest path to every point in the search tree it finds the single shortest path to the goal node.

Breathd first search is only optimal when all steps cost the same, because it always expands the shallowest unexpanded node. With a simple extension to breadth first search we can create an algorithm that is optimal for any cost. Instead of expanding the shallowest node like with BFS it instead expands the node, n, with the **lowest path cost**.

If all steps cost the same then Uniform Cost Search is identical to BFS.

Uniform cost search does not care about the number of steps a path has but instead it cares about their total cost. Therefore it is possible for BFS and Uniform Cost Search to get stuck in an infinite loop if it ever expands a node that has zero-cost leading back to the same state.

In other words, if Uniform Cost Search expands into a node with a cost path of zero from a node with a cost path of zero and there are two routes to each other it can get stuck in an infinite loop.

We can gurantee completeness using this search method by making sure the cost of every step is greater than or equal to a small positive constant.

Uniform Cost Search is guided by path costs and not lengths so it's complexity cannot easily be shown. Instead, let X be the cost of the optimal solution and assume that every action costs at least Y. Then the algorithms worst case scenario is O(b[X/Y]) which can be much greater than B^d, which makes Uniform Cost Search slower and more resource hungry than Breadth First Search.

Depth Limited Search

Depth Limited Search is a vartion on DFS whereby a limit is set to the depth of DFS so it does not go on forever or too long. It also solves the infinite path problem.

If the goal is at level N and we choose a depth of D and D < N then Depth Limited Search is incomplete.

The solution found is not guranteed to be the shortest optimal path.

The time complexity of Depth Limited Search is B^d where d is the depth limit.

The space complexity of depth limited search is B*d where d is the depth limit.

Iterative Deepening

Iterative deepning is a varation on Depth Limited Search whereby the depth limit is increased if the goal is not found. Iterative Deepning often starts at level 0 and then increases by a singular level if the goal isn't found on that level.

Iterative Deepning Depth First Search combinds the best parts of depth-first search and breadth first search.

Iterative Deepning's memory requirements are small, O(bd) where b is the amount of nodes generated and d is the depth level.

Iterative deepening may seem wasteful because nodes are generated multiple times, although this is not very costly. Consider a search tree with the same branching factor at each level; most of the nodes will be on the bottom level so it does not matter much to generate upper level nodes repeatedly.

In iterative deepning, nodes on the bottom level are generated once, those on the next to bottom level are generated twice and so on up to the children of the root node, which are generated d times.

Bidirectional Search

The idea of bidirectional search is to run two searches, one forward from the start state and one backward from the goal state, stopping when the two searches meet. The idea is that b^d/2 + b^d/2 is much smaller than b^d.

Bidirectional search works by having one or both of the searches check the next node to see if it is in the fringe (frontier) of the other search tree, if it is then a solution has been found.

The time complexity of Bidirectional search is O(B^d/2)

Bidirectional search must be able to calculate the predecessors of states and must also know where the goal is.

Informed Heuristic Searches

An informed search stratergy is one that uses problem-specific knowledge to find solutions more efficiently than an uninformed search strartergy.

Greedy Best-First Search

Greedy attempts to expand the node that is closet to the goal on the idea that it is likely to lead to a solution quickly.

It uses heuristics to determine which nodes are 'closet' to the goal node.

Greedy search is like depth first search in that it prefers to take a singular route to the goal and will back up when it hits a dead end. Greedy search is not optimal and it is incomplete because it can get stuck within an infinite loop.

The worst case time complexity of greedy search is O(b^m) where b is the amount of nodes and m is the maximum depth of the search space.

A* Search

The most widely known form of search algorihm is A* (a-star). It chooses its path based on 2 statistics, g(n) and h(n). g(n) is the cost to reach the node and h(n) is the heuristic function that shows the cost from the node to the goal.

f(n) = g(n) + h(n)

Provided that the heuristic function satisfies certain conditions then A* is both optimal and complete.

A* is optimal if h(n) is an admissable heuristic. That is provided that h(n) does not over estimate the cost to reach the goal. Admissable heuristics are often optimistic as they think the cost of solving the problem is less than it actually is. Since g(n) is the exact cost to reach n, we have an immedtiate consequence that f(n) never overestimates the true cost of a solution through n.

A* is normally not suitable for large search problems.

Games as Search Problems

In traditional search algorithms the user makes all the moves, however in a game we implement search algorithms against an unpredictable enemy.

The best possible way is to calcualate every single possible move the enemy can make and counter them, although this will cause a combinatoral explosion in which the computer will take too long to calculate the correct answer.

In reality, we need to use heuristics to calculate search problems in games.

In some games we have a fully observable environment, we know everything about the environment such as Chess or Go. In other games we have partial information where we don't fully know the environment, such as Poker.

Minimax Algorithm

n a game, Min and Max are two players. Max wants to win (maximise the utility of each move) whereas Min wants Max to lose (minimise utility for max).

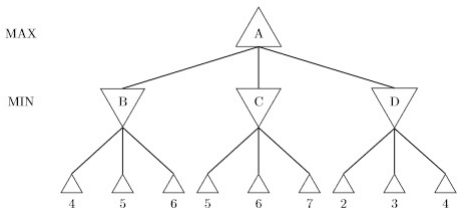
The game Minimax is used on has to be a zero sum game, that means that there can only be winners and losers, nothing in between; like Chess.

The set of goal states is replaced by a utility function.

Max wants a strategy for maximising utility assuming min will do the best it can to minimise max's utility.

The utility of a MAX-state (where Max moves) is the maximum of the utilities of its successor states.

The utility of a MIN-state (where Min moves) is the maximum of the utilities of its successor state.



The Minimax starts from the bottom, with Min (player 1) starting first. You know min starts first because it is denoted in the left hand side. Min would choose the lowest possible value, so for level 1 it would be {4, 5, 2} since these are the lowest possible values in the tree.

We then move one level up. The Minimax values of A is the maximum value of {4, 5, 2} which is 5. So Max would choose 5.

In game playing, minimum would want Max to choose the one with the lowest utility, as a higher utility might attack or hurt player 1 more than a lower utility.

In a game such as chess the trees are extremely large. In chess the number of moves grow exponentially, after 4 moves there are 288 billion different possible positions. This is where Alpha-Beta pruning comes in. We cut off large chunks of the tree that we believe we no longer need in order to allow the Minimax algorithm to run in efficient time, that is time that will take less than 10 minutes to run (roughly).

Minimax tries to reduce the maximum damage the opponent can do whilest reaping the maximum possible reward for us assuming the opponent plays optimally.

There is also a second list, a list of every node that has already been visited in the search tree.

If you enjoyed this article, connect with me to learn more like this 😃

LinkedIn | Website | Twitter upscribe

linkedlin twitter githib

Don't like this article or want to add something? Submit a pull request here:

Links to twitter, github, etc link to upscribe, paypal.me, ko-fi Upscribe link

Previous articles

Link to github repo

Link to PDF