CSC A48 – Unit 5 – Graphs and Recursion

1.- Expanding our problem solving toolkit

Up to this point, we have a pretty good handle on how to store, organize, and access data. We have learned about computational complexity, and how we can use complexity analysis to gain a better understanding of different problems, and of the algorithms we implement to solve them. We have in our toolkit a couple of very useful ADTs, and we have spent a good amount of time practicing how and where to use them.

In this unit, we will add to our toolkit two of the most general and powerful tools for problem solving in computer science: Graphs, and recursion.

Graphs are used to model all kinds of real-world items, and real world problems, where the key to understanding the particular problem is the relationship between items in our collection. Recursion gives us the ability to understand, manipulate, and solve problems whose particular properties make regular processing with loops and conditionals very difficult.

Together, graphs and recursion will open the door for you to work on a variety of fascinating applications in all fields of knowledge. So let's dive in and find out what we can do with these two tools!

2.- Graphs

In Computer Science, graphs are used as a model to represent items of interest, where the ***relationship*** between items is relevant to the representation of our collection and to a problem we wish to solve. To understand this, let's take a look at the type of information we have been working with up to this point, and the kinds of problems we have been solving with the tools we have acquired up to this point.

So far, we can work with (possibly very large) collections of data items, these items can be regular C types, or compound data types which contain multiple fields. However, one key property of the data we have been working with, and the problems we have been looking at, is that each item is fundamentally independent of each other.

For example, we have been working with restaurant reviews – we can search for specific restaurants, find out which restaurants have a review score above a specific value, check out the restaurant addresses, and so on. *Importantly: Each review is processed independently of the rest, and the problems we have been solving do not require us to model in any way the possible relationships between reviews for different restaurants.*

However – relationships between data items are ***extremely important!*** In the case of our restaurant reviews for instance, we may want to consider that different locations of the same food chain are related to each other: They serve basically the same food! So we should expect their scores to be similar to a large degree. Restaurants offering a particular type of food (e.g. Mexican tacos) are related,

and comparing their reviews is informative. Restaurants in the same part of town are also related (geographically), and this is an additional source of information that we haven't used thus far: Perhaps users in a particular part of town are more 'picky' and like to give worse reviews regardless of how good a restaurant is, or perhaps they just enjoy more certain kinds of food. With what we have learned up to this point, we can't really begin to answer questions of this type.

Graphs provide us with a way to ***model, reason about, and manipulate*** data items and *the relationships between them*. With graphs, we can ask questions such as 'what kinds of restaurants do my friends like?', 'what are the best courses people I know take in the 2nd year?', 'How good are movies directed by Martin Scorcese?', and 'What is the fastest route to get from UTSC to the wonderful pizza place I love best?'

Graphs encode data items, and their relationships or interactions, in a way that allows us to implement algorithms that explore the structure of the data we are studying, and that allow us to find meaningful interactions between data items, and to look for interesting patterns in complex data sets.

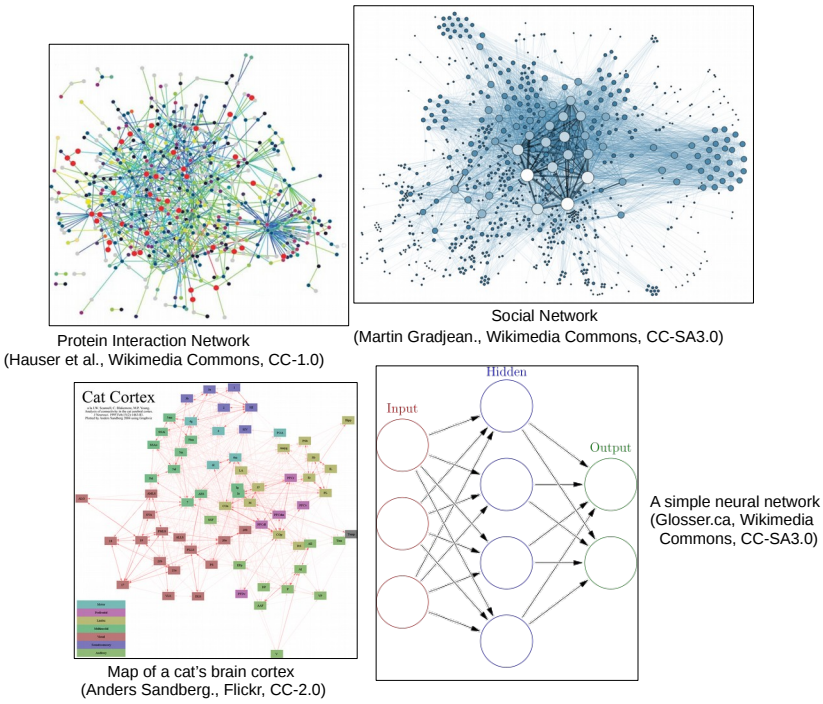
In Computer Science, graphs are used to represent information in ***networks of all types:***

- Social networks
- Transportation
- Genomics and bioinformatics
- Computer networks and the Internet
- Courses and their pre-requisites

And they are essential for handling problems such as:

- Natural language processing and understanding (translation, answering user questions)
- Image understanding
- Scheduling and optimization of real-world operations (airports, manufacturing, etc.)
- Artificial Intelligence (used in search, game playing, decision making, neural networks)
- Path planning (if you use Waze or Google drive to get to places, those run on graphs!)
- Data modeling and visualization

… and an whole world of other fascinating applications!



Drawing 1: A handful of examples of applications of graphs

3.- Definition of a Graph

A graph consists of:

- A set of ***nodes*** corresponding to ***data items we are working with***. The set of nodes is usually called ***V***. In books, lecture notes, and future courses, you may find the term ***vertex*** is used instead of ***node***. They are the same thing.

- A set of ***edges*** which are the ***connections between nodes***, and represent ***the relationships*** existing between ***data items in our collection***. The set of edges is usually called ***E***.

Together, they define the graph G=(V,E).

A sample of a graph is shown in the picture below

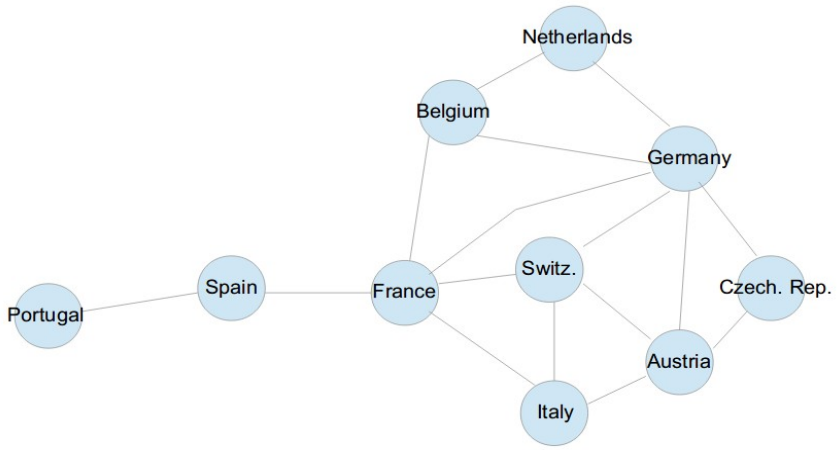


*Figure 1: A graph representing countries in western Europe. Each node corresponds to one country (so the set of nodes **V** contains one node per country, labeled with the country's name). Edges are used here to indicate two countries have a shared border (so the set **E** of edges will contain one entry for every shared border between two countries).*

The meaning of the ***nodes*** and ***edges*** depends on the particular problem we are studying. For example:

- In a social network, ***nodes*** correspond to people, and ***edges*** represent the fact that two people know each other (are friends, colleagues, etc.).
- For a mapping application, ***nodes*** correspond to locations, and ***edges*** correspond to streets linking locations together.
- For neural networks, ***nodes*** correspond to processing elements (neurons), and ***edges*** represent the flow of information within the network.

As you can see, graphs are a ***very general*** way of representing information and relationships between items. You can build a graph for pretty much any problem you can think of, as long as you can find some meaningful way in which data items for that particular problem relate to each other.

You have already been working with graphs! Trees, such as ***BST***s are graphs (where nodes in the tree are related to each other by parent-child relationships), ***linked-lists*** are also graphs (nodes are related to each other by a predecessor-successor relationship). So in fact, you already have plenty of experience working with information stored in graphs.

Types of Graphs

There are ***two general types of graphs*** we will use widely:

Un-directed graphs: Like the example shown above, where edges between nodes are ***shared***, and the relationship between the nodes ***goes both ways.*** In the example above, if Portugal shares a border with Spain, it ***must be true*** that Spain shares a border with Portugal.

Directed graphs: In this type of graph, edges have ***a direction***, the relationship between the nodes goes ***one way.*** Examples of these are trees like the ***BST***s you've been working with: Edges in ***BST***s go from ***parent to child***. If node ***a*** is a parent to node ***b***, the reverse ***can not be true***. An example of a directed graph is shown below:

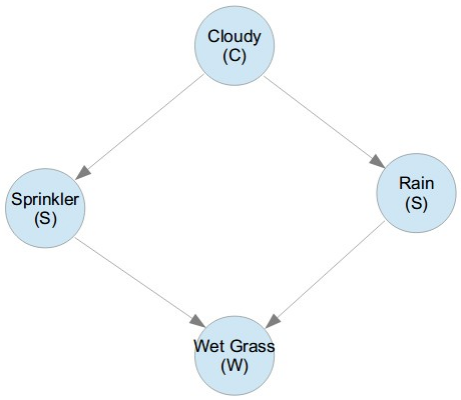


*Figure 2: A sample of a directed graph. This one represents variables used to model an inference problem: An observer sees the grass is wet, and is trying to determine the reason why. There are four variables (hence, four nodes in **V**), corresponding to the grass being wet (true/false), it having rained (true/false), sprinklers being on (true/false), and the sky being cloudy (true/false). **Directed edges** indicate that **the parent variable's value affects the value of the child variable**, e.g. If we know that rain=true, that will affect the value of wet grass.*

What type of graph we use depends on the data we are working with, and the problem we are trying to solve.

Important terms to remember related to graphs

- ***Neighbours***: A node ***v*** is a neighbour of node ***u*** if (for ***un-directed graphs***) there exists an edge $\{u, v\} \in E$ joining both nodes. For ***directed graphs***, a node ***v*** is an ***out-neighbour*** of node ***u*** if there is an edge $(u, v) \in E$ from ***u*** to ***v***. Conversely, ***v*** is an ***in-neighbour*** of ***u*** if there is an edge $(v, u) \in E$ from ***v*** to ***u***.

- ***Neighbourhood****:* For ***un-directed graphs***, the neighbourhood of node ***u*** is the set of all nodes that are neighbours of ***u***. For ***directed graphs***, there is an ***out-neighbourhood*** and an ***in-neughbourhood***, corresponding to nodes that are connected to ***u*** by edges ***leaving u***, and nodes which ***u*** is connected to by edges ***arriving at u*** respectively.

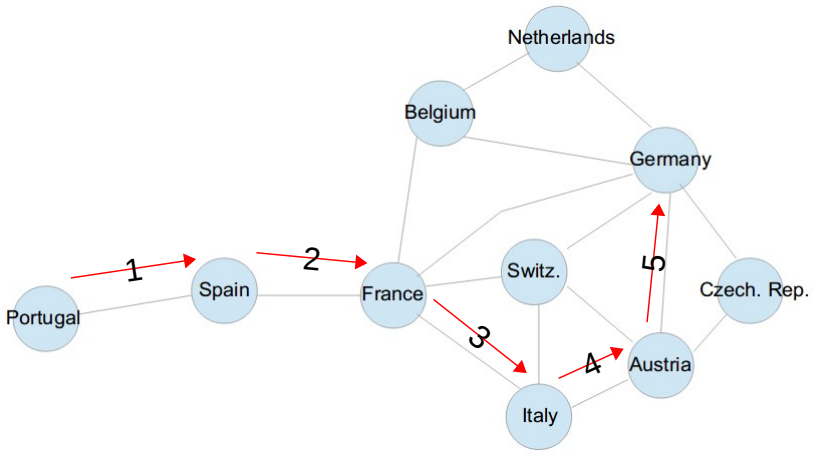
- ***Degree***: For ***un-directed graphs*** the degree of a node ***u*** is the size (number of nodes) in the neighbourhood of ***u***. For ***directed*** graphs we have an equivalent ***out-degree***, and ***in-degree***.

Exercise: For the graph representing the countries of Europe (see the figures above), what is the

neighbourhood for the 'Germany' node? What is the degree of this node?

Exercise: For the graph above representing the problem of figuring out why the grass is wet, what is the ***out-neighbourhood*** of the node for 'Cloudy'? What is the ***out-degree*** for this node?
What are the ***in-neighbourhood***, and ***in-degree*** for this node?

- ***Path***: A path through a graph is ***a sequence of consecutive nodes*** that can be ***visited*** by following existing edges between each pair of consecutive nodes. For example, in the graph below, there is a ***path*** from ***Portugal*** to ***Germany*** that visits in sequence *Portugal → Spain → France → Italy → Austria → Germany.*
(incidentally, that would probably be an amazing trip to make!)



Note: For ***un-directed*** graphs, we can travel along edges in either direction. But for ***directed*** graphs, we can only go from node ***u*** to node ***v*** if an edge exists that goes from ***u*** to ***v***.

- ***Cycles***: For ***un-directed*** graphs, a cycle is a path with at least 3 nodes that starts and ends at the same node. For ***directed*** graphs, a cycle is a path that begins and ends at the same node, but in this case the path can have any number of nodes.

A few awesome applications of graphs

1) Network modeling: This includes ***any*** kind of network, including ***the Internet***, ***social networks, professional networks, the electricity grid, a city's network of streets, protein interaction networks, transportation networks,*** etc. Graphs can be used in such networks for: ***Analysis of structure, simulation, detection of points of failure, route planning, relevance scoring,*** and many other applications.

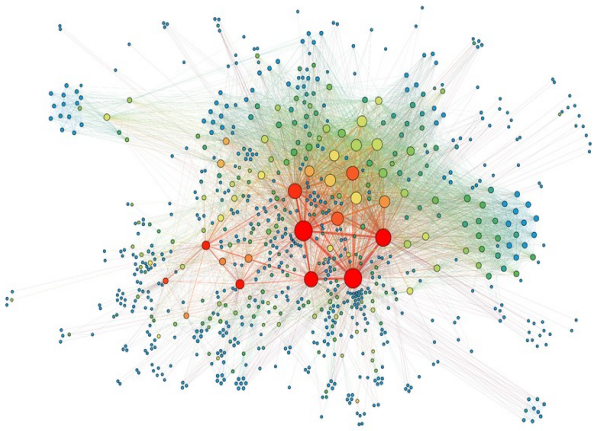
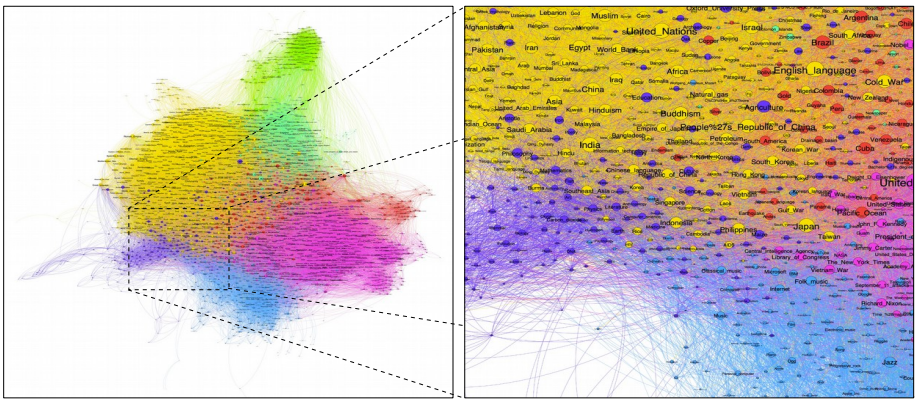


Figure 3: Social network analysis. Each circle is a node representing a person, the colour and size of the node indicate how important each node is in the graph - this is related to how well connected the node is, and whether its neighbours are themselves important.

2) Document analysis: Representing a collection of documents. These can be ***text***, such as articles, books, on-line blogs, tweets, etc.), or any other kind of media such as ***images, music and sound recordings, video,*** etc. Documents can be linked in a number of ways, for instance, they can be linked by ***source*** (who generated the document), by ***type*** (e.g. linking together newspaper articles, separately from book chapters, journal papers, etc.), by ***topic*** (for instance, linking together all music videos of classical music), or by any other property or combination of properties that is relevant to the problem we are studying. Once the graph has been created, we can use it to: ***cluster documents*** (group them by a relevant property), ***determine relevance*** (which documents are more commonly accessed or referenced), ***and discover structure*** in the collection. This forms the basis of ***recommendation systems*** used to determine what a user is likely to be interested in.



Drawing 2: Top 2500 Wikipedia pages, grouped by similarity of content. Note that colour corresponds loosely to topic. Image by Matt Biddulph, Flickr, CC-SA2.0

3) Numerical simulation: Many applications in ***Physics, Engineering, Medicine, Weather***

Analysis, Computer Aided Design, and ***Computer Graphics*** rely on numerical simulations performed on a ***mesh decomposition*** of a surface or volume – that is, ***nodes are placed at pre-defined locations on the surface or inside the volume***, and then these nodes are ***linked to form a mesh***. The values of interest for the simulation (e.g. wind speed in a weather model for a storm) are computed at each node, and information is propagated via the edges linking neighbouring nodes to carry out the desired simulation.

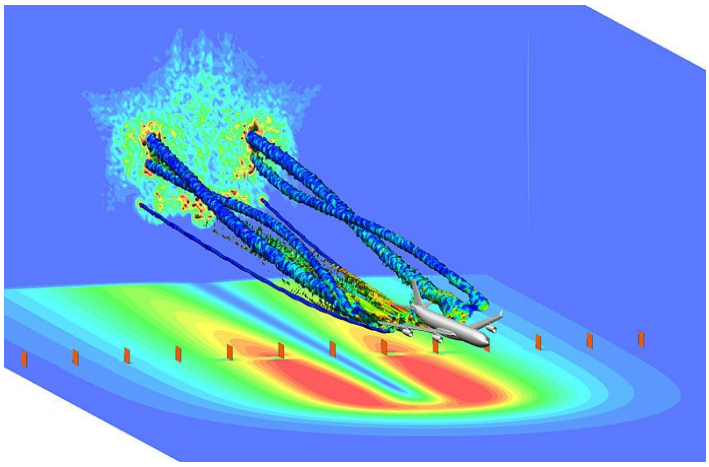


Figure 4: Simulation of the turbulence generated by an A340. A mesh of nodes (not visible in the image) distributed over the volume of this simulation forms the basis of the computation. The same mesh is used for visualization, by assigning a colour to nodes of interest. Image: Deutsches Zentrum für Luft und Raumfahrt, Wikimedia Commons, CC-By3.0

4) Artificial Intelligence. A significant number of important applications in AI (and hence in Machine Learning) rely on graphs for representing information and for carrying out relevant processing. A very small sample of the kind of problems you can solve with graphs in AI include: ***path planning and route-finding, constraint satisfaction (scheduling and industrial process optimization), game playing (this has serious applications in finance, advertising, etc.), inference and decision making under uncertainty, and the current and very promising field of Deep Learning and its applications.***

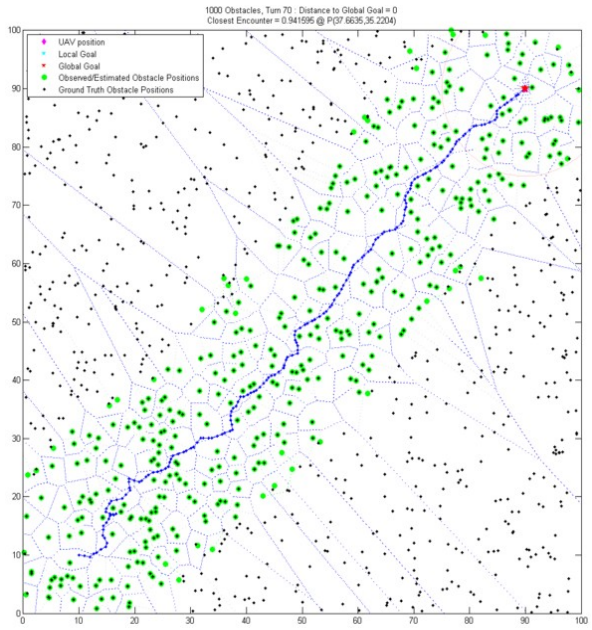


Figure 5: A path-planning simulation for an autonomous flying vehicle carrying a search and rescue mission in a forest. Image: Elucidation, Wikimedia Commons, CC-SA3.0

It should be clear to you that graphs have an amazingly wide range of applications, and there is a variety of courses in computer science and other scientific disciplines in which you can learn as much as you like about particular problems you are interested in. Here in A48, our goal will be to understand the fundamental concepts related to storing, manipulating, and using graphs defined over collections of data. What you learn here will be the basis for later understanding specialized material in almost all areas of application of computer science.

3.- Representing graphs

There are two main ways of representing graphs (both un-directed and directed). Each has its own advantages and disadvantages, and the choice of method for any particular application will depend on how the graph will be used.

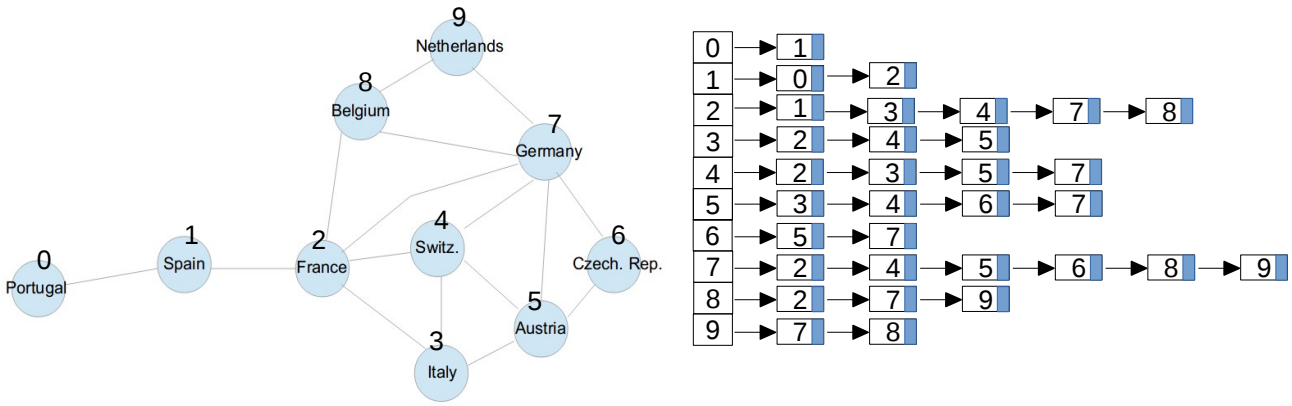
First, the ***nodes***: These correspond to data items in our problem, and can be stored using any of the data structures we have studied up to this point (e.g. arrays, linked lists, trees, etc.). You know that in terms of choosing what data structure to use to store your collection, the usual considerations apply!

That leaves the problem of storing the set ***E*** of ***edges*** for the graph. We need a way to keep track of which nodes are connected, and in the case of directional graphs, what the direction of each edge is.

The two most common ways of doing this are:

1.- Adjacency list – The adjacency list is an ***array*** with ***one entry per node***. The i^{th} entry in the array contains a pointer to ***a linked list*** that stores the indexes of nodes to

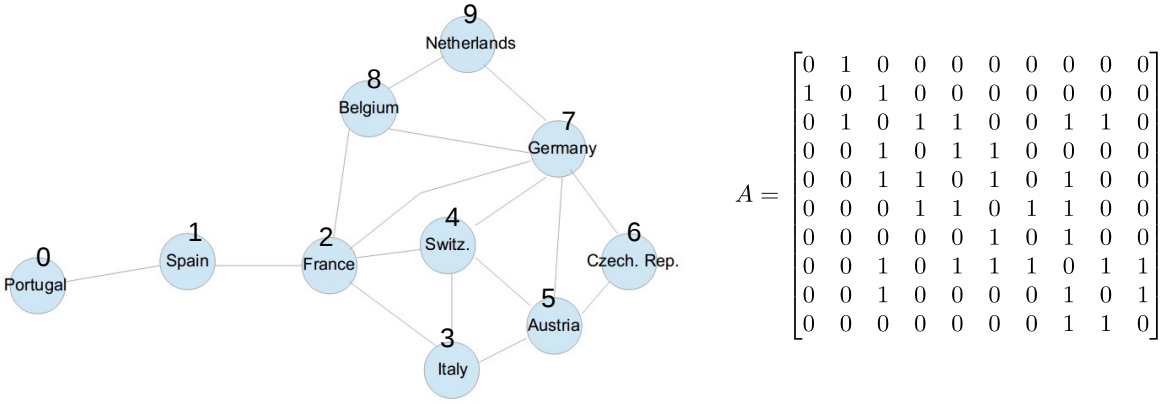
which node *i* is connected. This is illustrated below



Drawing 3: An adjacency list for the European countries graph. There is one entry per node, and each entry points to a linked list containing the indexes of the neighbours for that node

Adjacency lists have the advantage of being space efficient – if the graph has a large number of nodes, but each node is connected to at most a few neighbours, then the ***adjacency list*** stores the required edge information in a very compact format – without wasting memory. Conversely, common graph operations such as ***querying an edge*** (figuring out whether two nodes are connected) requires list traversal, which as we know can be slow.

2.- Adjacency Matrix – As the name implies, the ***adjacency matrix*** is a 2D array of size ***N x N,*** where ***N*** is the number of ***nodes*** in the graph. For ***un-directed graphs,*** The entry ***A[i][j]*** is ***1*** if nodes ***i*** and ***j*** are connected, and ***zero*** otherwise. For ***directed*** graphs, entry ***A[i][j]*** is set to ***1*** if there is an edge ***from i to j***, and is ***zero otherwise.***



Adjacency matrices have the same advantages and disadvantages of arrays: Edge queries now have no overhead, unlike linked lists which require list traversal. Adding or deleting edges, and finding out whether two nodes are connected requires a single access to the matrix. Conversely, they are not space efficient. Even in the small example above, you can see that the majority of the entries in the

matrix are zero. For a very large graph, the adjacency matrix will waste a significant amount of space – and may in fact not fit in memory!

Note: For ***un-directed graphs*** the ***adjacency matrix*** must be symmetric.

Complexity of fundamental operations on Graphs

The choice of representation we use for the graph (adjacency matrix or adjacency list) will have implications in terms of the computational cost of a few common graph operations:

- ***Edge query:*** Finding out whether two nodes (u,v) are connected
  Adj. List: $O(|V|)$ Adj. Matrix: $O(1)$

- ***Inserting a node:*** Adding a new node to the graph
  Adj. List: $O(1)^*$ Adj. Matrix: $O(|V|^2)$

- ***Removing a node:***
  Adj. List: $O(|E|)$ Adj. Matrix: $O(|V|^2)$

- ***Inserting an edge:***
  Adj. List: $O(1)^+$ Adj. Matrix: $O(1)$

- ***Removing an edge:***
  Adj. List: $O(|V|)$ Adj. Matrix: $O(1)$

Notes:

** Assuming there's space in the array of pointers to linked lists of edges, or that we are storing these pointers in a linked list*

+ Assuming the new edge is inserted at the head of the linked list of edges for a node

The complexity class ***O(1)*** is something we haven't seen before - it represents an operation whose computational cost is constant (***it doesn't mean it is exactly 1***). That means, it does not depend in any way on ***N***.

In the results above, $N=|V|$, the number of nodes in the graph, and $M=|E|$ is the number of edges in the graph. The results above are expected from the way the adjacency list and the adjacency matrix work, the complexity result for adding a new node to the adjacency matrix is a result of the fact that the matrix has size $|V|*|V|$, and the whole matrix has to be re-allocated and copied over with an extra row and column.

Exercise: For the graph shown below

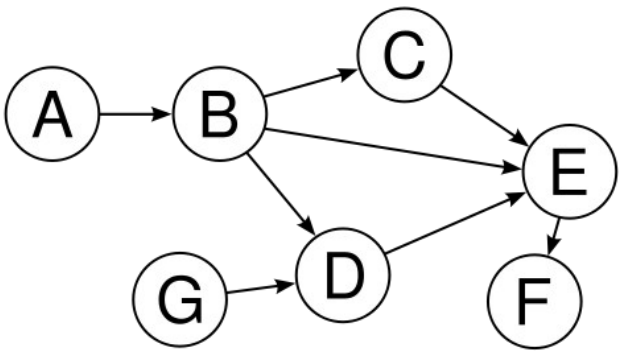


Figure 6: A small directed graph. Image: David W., Wikimedia Commons, Public Domain

- Show the adjacency list, use indexes 0 through 6, starting with A at index 0.
- Show the adjacency matrix for the graph.

Note: Here we have discussed graphs where edges are either there, or not. The adjacency matrix has a 1 or a 0 to represent this. More general graph problems may require edges to be ***weighted***, that is, they have a value attached to them that represents information about the problem (e.g. for a graph linking city intersections with roads, the edge weight may represent the speed limit on the corresponding road). The representation is the same for these weighted graphs, except that now the adjacency matrix has real values for the weights of edges, and zero only when there is no edge. An adjacency list would have to store the weight of each edge.

Solving problems with Graphs!

Now that we know how to represent and store a graph, we're ready to start thinking about the kinds of problems we can solve using graphs. However, before we can really explore interesting applications of graphs, we first need to study a general problem solving technique that is very strongly tied to graphs, and to algorithms that process information encoded in graphs: ***Recursion***.

5.- Recursion as a tool for problem solving

The first thing to point out with regard to recursion ***is that you already know it, and have been using it even if we didn't call it by its name at the time!*** Recall we made a point of noting that both ***linked-lists*** and ***BST***s are graphs. And we just said that recursion is strongly tied to algorithms that work on graphs.

Well, as it turns out, all the operations we defined on ***BSTs*** in the previous section are recursive by nature. Consider the insertion of a new node into a ***BST***: Start with the ***root node for the tree***, check if it's ***NULL***, and if not, decide whether the node should be inserted on the *left* or *right* subtree. Then

recursively insert the node on the correct subtree. We didn't call it *recursion* at the time, but the process intuitively made sense from looking at the structure of the tree and thinking about how the insertion process had to work.

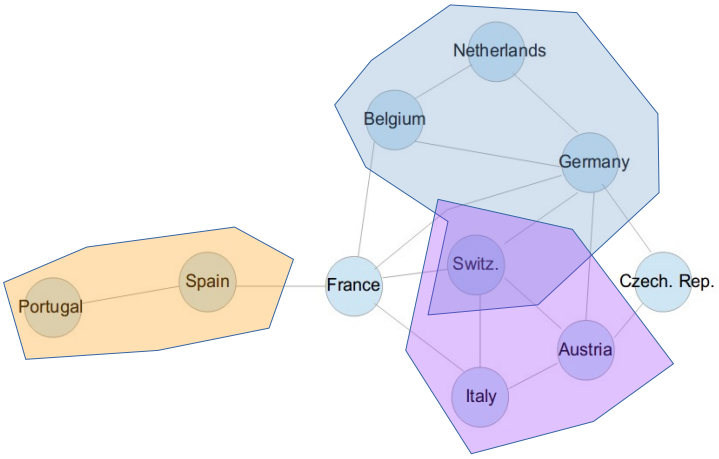
The same applies to ***BST*** search, deletion, and tree traversals! All of them are recursive in nature and they are this way because of the structure of the tree – remember we noted that every sub-tree of a ***BST*** is also a ***BST!***

So you already know of at least one data structure, and the algorithms that we implemented to make it work, that is *recursive* in nature, and that has to be *as a result*, processed *recursively*.

What we need to do here is acquire a more general picture of what recursion is: ***a way of thinking about problems, a way to implement code that solves particular problems, and a tool for managing data whose structure is itself recursive.***

Examples of Recursive Problems and Data Structures

Graphs are a recursive data structure: Every sub-graph of a graph is also a graph:



*Drawing 4: A graph and several possible sub-graphs (in colour), each subgraph is itself a graph with its sets **G** of nodes and **E** of edges.*

We had already seen this property with ***BST***s, but now we know it's a general property of graphs. Linked-lists, which are also graphs, are also recursive in nature: Every sub-list of a linked-list is also a linked-list.

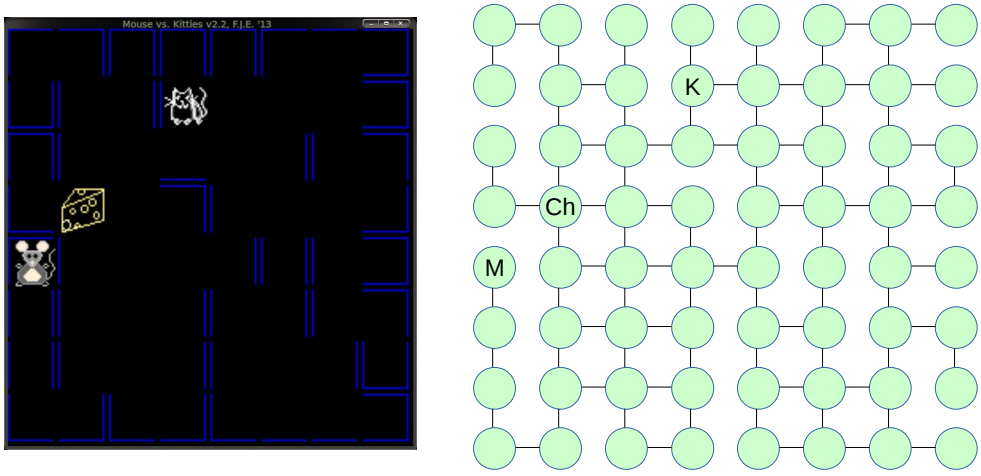
And why is that interesting?

Because we can take advantage of the recursive structure of the graph to solve a seemingly complex problem by:

Taking the original input graph
Breaking it up into smaller subgraphs
Breaking those up into even smaller subgraphs
And so on…
Until the subgraphs are so tiny solving the problem is trivial
We use that solution to solve the slightly larger subgraphs
And then the even larger subgraphs
And then even larger subgraphs
Until we have the solution to the original problem!

This general process for working on graphs is an example of ***recursion***, and variations of the algorithm above are used for ***path finding, robot planning, scheduling, image pattern detection***, and many other applications. Let's see an example in ***path planning***. The setup is as follows: ***any map*** whether it represents city streets, network routers available within some region, or in the case below, a little maze, can be represented by a graph with ***one node per location***, and ***edges linking together locations that are connected.***

In the case of city maps, the ***nodes*** can represent street intersections, and the ***edges*** can represent streets linking intersections together (***question:*** would this be a ***directed graph*** or an ***un-directed graph***?). For computer networks, the ***nodes*** would correspond to routers through which network traffic can flow, and the ***edges*** would correspond to data links between routers. In the example below, we have an 8x8 map, each location has been assigned a node in a graph, and the nodes are connected if the corresponding map locations are connected (there are no walls in between them).



Drawing 5: A sample of a graph representing a small map. Each node corresponds to a location in the map, edges link together locations that are connected (no walls between them), and represent the fact that someone walking around this map could move from one location to another if the corresponding nodes in the graph are linked.

Problem: How do we find a path from the mouse to the cheese? In terms of our graph this means finding a path from the node labeled '*M*' to the node labeled '*Ch*'. For you, looking at the graph above, it's very easy to see the path the mouse should take. For a computer working on a graph this is not so simple.

Exercise: Write a program that uses ***loops*** (no recursion!) to solve the path finding problem above.

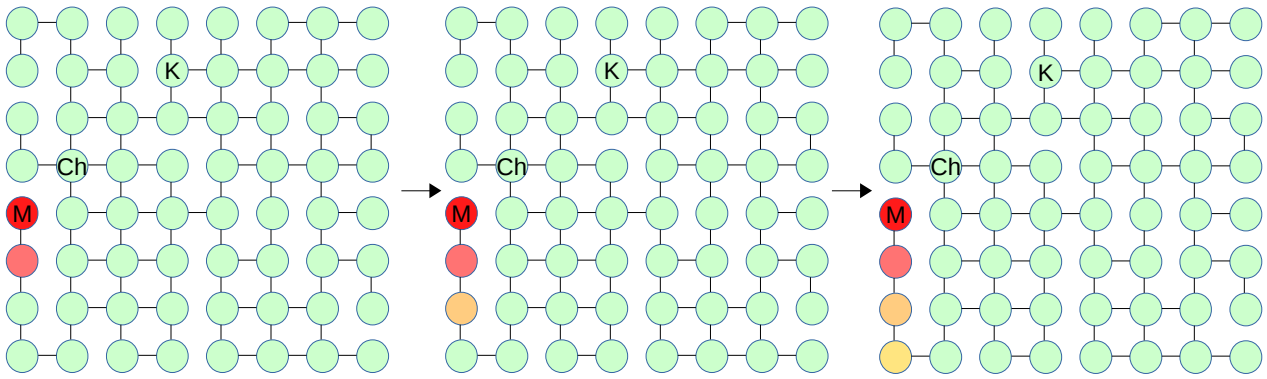
If you spend any time at all solving the exercise above, you'll quickly realize that using loops on graphs quickly leads to code that is (at best) long, cumbersome, full of special cases, and that will not work well on slightly different versions of the problem above. *Since graphs are a recursive data structure by nature, you should expect that non-recursive algorithms working on graphs will be cumbersome and difficult to implement, test, and maintain.*

Let's see what happens if we try to solve the problem above ***recursively***, by breaking our original path-finding problem into smaller (simpler) ones until the solution is trivial.

Our path-finding problem begins with our ***initial location***. In the example above, it's the node where the mouse is at. Then, finding the path is just a matter of applying the following process:

find path to cheese:
Ask my neighbours: What is the path from ***you*** to cheese
They ask *their* neighbours: What is the path from ***you*** to cheese
They ask *their* neighbours ….
… until – one of the neighbours-neighbours...neighbours is the cheese!
Now one of the cheese's neighbour knows where to go
Then a cheese's neighbour's neighbour knows where to go
… until, the mouse knows where to go!

Let's see this working on the graph above:

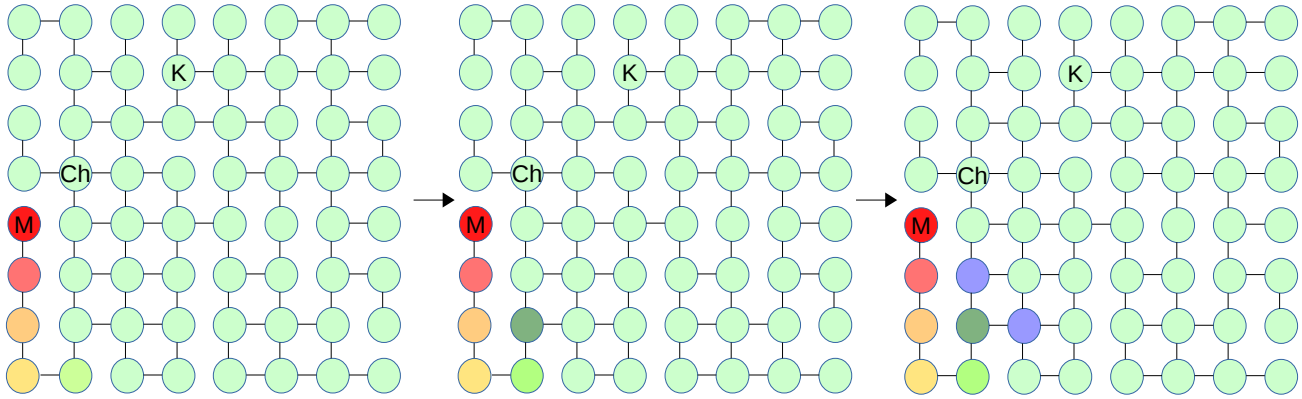


Starting at the mouse location, ask neighbours '*what is the path to the cheese*'?

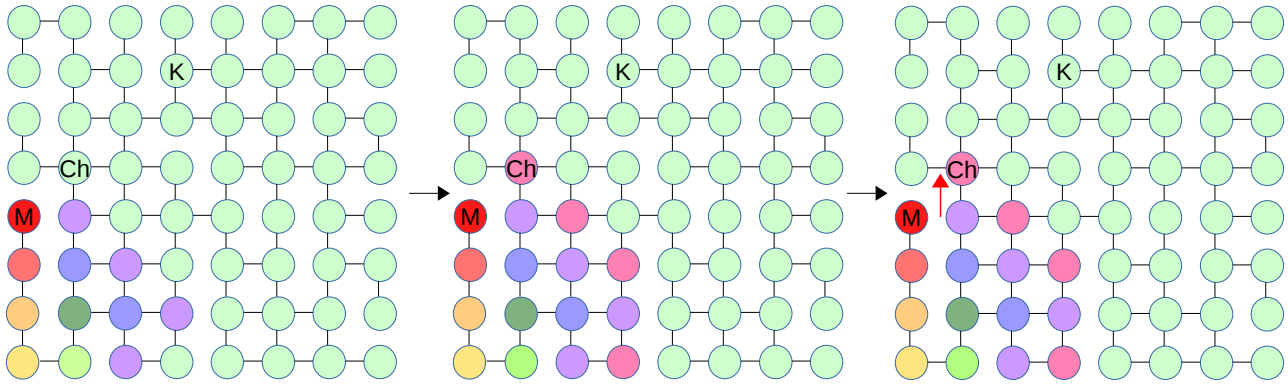
Then *they* ask *their neighbours*: '*what is the path to the cheese*'? (not including nodes we've asked already)

Then *they* ask *their neighbours*: '*what is the path to the cheese*'?

keep going…



and going…

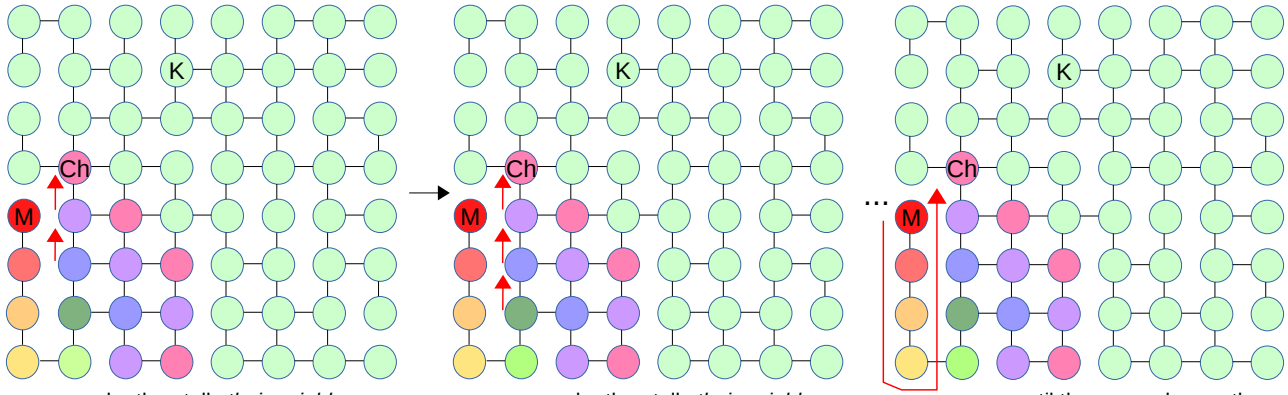


Each successive ***step***, neighbours Further out are asked (marked by colour)

Until one of them is the node we want!

That node tells its neighbour ***where to go***

we found it! Pass the location back, neighbour to neighbour, until it gets to the mouse!



… who then tells *their neighbour* ***where to go***

… who then tells *their neighbour* ***where to go***

… until the mouse knows the ***path to the cheese***

Notes: We just carried out a ***recursive*** algorithm! It keeps asking neighbours for information until we find one that knows what the answer is, then the information is passed back. ***This is an essential part of recursive problem solving:***

– The problem is progressively broken down, made smaller, or otherwise simplified at each step.
- Then information needed to form the solution is passed ***back*** one step at a time, until the solution for the original problem is obtained.

We will discuss these two components in more detail. For now, it's enough to remark that the process we just carried out is called ***graph search***, and is the foundation of ***path planning***. If you have ever used ***Google Maps,*** or ***Waze*** to find your way around, they are solving the same problem, using slightly more advanced ***graph search implementations***, but fundamentally doing the same thing we did in the illustrations above. We will come back to graph search later on, and you'll get to implement it! Before then, let's look at a few more ***commonly found*** recursive problems to strengthen your understanding of what recursion is all about.

Note: If you're curious about the many applications of graphs in Artificial Intelligence, including more advanced methods for path finding and planning, don't forget to check the AI course D84!

Divide-and-Conquer methods

Divide-and-conquer methods are behind some of the most powerful tools we can find in computer science – including *binary search*, *quicksort*, the *Fast Fourier Transform* (used extensively in signal processing and signal analysis), *mergesort* (a sorting algorithm with a guaranteed worst-case-complexity of ***O(N Log(N))***, and the *binary space-partitioning trees* for determining object visibility in computer graphics.

They are *naturally recursive* in that by definition they work by splitting a problem into smaller and smaller instances, applying to each of these the same algorithm, until the problem is easily solvable; then combining solutions for smaller instances of the problem to build the solution to larger and larger instances all the way back to the original one.

Example: *Mergesort* is a sorting algorithm guaranteed to sort an input collection in ***O(N Log(N))*** time. The method works as follows (pseudocode):

```
mergesort(input_array)

        if the length of input_array is <= 1, then array is sorted: return input_array

        else
                split array into 2 sub-arrays: lower_half, and upper_half
                sorted_low = mergesort(lower_half)
                sorted_high = mergesort(upper_half)

                Merge in sorted order the two sub-arrays sorted_low and sorted_high
```

```
                    to build the complete sorted_array

                    return sorted_array
```

It is likely you will do a full analysis of the complexity of *mergesort* next year in ***B63***, for now, you can obtain an intuitive understanding of why it has a complexity of ***O(N Log(N))*** by considering the following:

- It takes $Log_2(N)$ splits to reduce the input array to sub-arrays of size 1 or 0 ***no copying of input data is needed –*** the splitting can be done by passing indexes into the input array indicating the first and last entries in a sub-array.
- Merging in order two sorted collections with ***N/2*** entries is done in ***N*** steps. We have to merge arrays up to $Log_2(N)$ times (for each level of splitting). Hence $N\ Log_2(N)$. Note that this step requires setting up a separate array to hold the sorted result as it is being built by the merge step.

The above is just a quick-and-dirty overview. As noted, the proper analysis of *mergesort* is left for B63, where you will look at the complexity of the algorithm both in terms of time, and in terms of the storage space it requires.

Computer Graphics

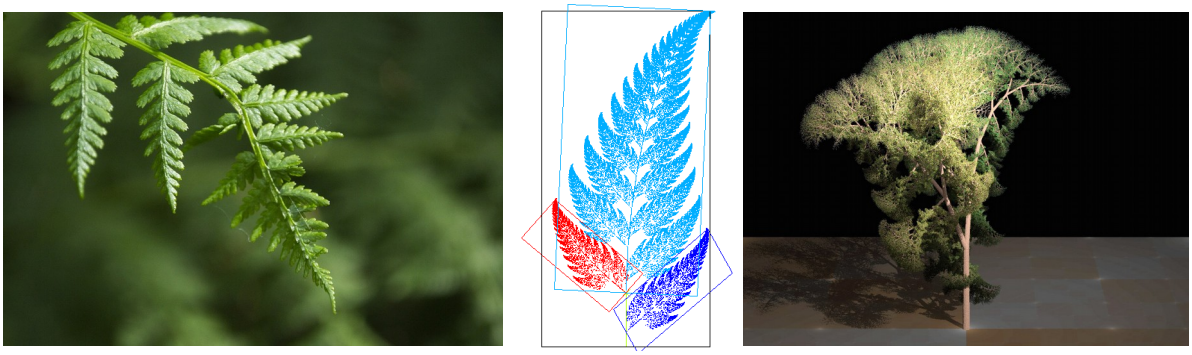
Recursive structures are common in computer graphics:

- Objects composed of parts (e.g. the various parts of a person or animal, plants, buildings) are often modeled using tree-structures – hence, graphs: recursive structure.

- The rendering (drawing) process for certain plants like ferns and trees is naturally recursive

- Animation of objects routinely requires us to recursively apply *transformations* (changing the shape, size, orientation, or some other property of the object) to objects and their parts and sub parts (for example: To animate an arm, we move the upper arm, then the lower arm moves relative to the upper arm, then the hand moves relative to the lower-arm, and so on).

- The most advanced rendering methods (capable of creating movie-quality, photo-realistic scenes) are naturally recursive. They rely on tracing the path of light as it bounces from one object to another, then another, then another, and so on until the entire path of light from a light-emitting object to the camera has been found



Drawing 6: A real fern (left), a computer model of a fern (center) showing how the fern's parts are just smaller versions of the whole shape, and a computer rendering (right) of a tree that uses recursive shapes, a tree-based representation, and a recursive rendering algorithm to create a life-like virtual plant. Images: (left) Pixbay, used with permission, (center) A. M. Campos, Wikimedia Commons, public domain, (right) Solkoll, Wikimedia Commons, public domain

From the above it should be clear that recursion is a fundamental tool in computer graphics. It is used in some way for almost every component of the process of defining, representing, storing, and rendering computer generated images.

Note: To learn more about how photo-realistic computer graphics are generated, check out the computer graphics course D18.

Programming Languages

If you plan to be a serious software developer, you will need to learn different programming paradigms. C is based on the *imperative* programming model – program statements change the value of data and the state of the program in order to achieve the program's goal. Many other programming languages work in basically the same way.

However, there is a different programming model called *functional* programming (***note that this doesn't have anything to do with using functions! The term has a different meaning here!***). Functional programming is built on the concept of a program being the result of the evaluation of a set of mathematical expressions or functions. Functional programming languages include *LISP* and its derivatives (*Scheme*, *Racket*), as well as *Haskell*. And they heavily rely on *recursion* for carrying out their work.

Note: You can learn about *functional programming* in the programming languages course, C24.

File system organization

As a final example of recursive problems and data structures, consider the structure of the file system in your computer. The information you have there is organized into ***folders***, each of which will contain multiple items which themselves can be ***folders***. The structure of the directories in the file system is in effect a tree (graphs again!), and is recursive.

Example: Finding a file matching a particular name in your computer:

```
findFile(directory, file name)
        if a file matching file_name is in directory
                return directory
        else
                for every sub_directory in directory
                        return findFile(sub_directory, file_name)
```

In effect, this process travels through the directory structure in your computer, from the top-level directory (My Computer, if you're in Windows, '/' in Linux), looking for the file. For any given directory, if the file is not in that directory it then checks each of the sub directories recursively (and you should by now understand that this means that for each sub-directory the same process will be carried out!) until either the file is found, or the entire directory structure in your computer has been searched.

The recursive structure of the file system is not limited to directories and files. For example, in Linux, the actual ***data blocks*** that make up a file are organized into a tree-structure in an ***inode*** (this has nothing to do with a certain company that sells phones, tablets, and computers):

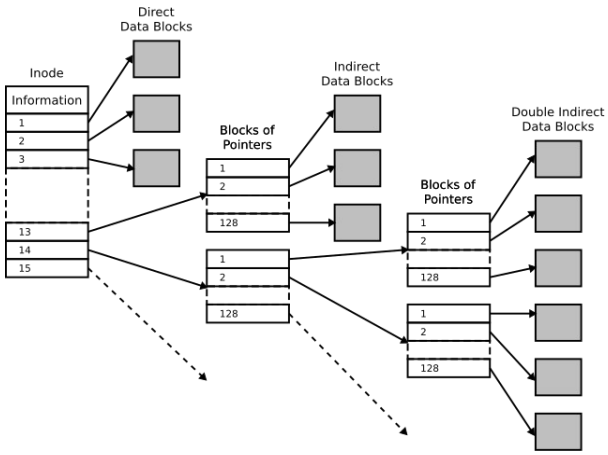


Figure 7: Structure of the data component of a Linux EXT2 File System inode. Image: timtjtim, Wikimedia Commons, CC-SA4.0

Note: To understand the principles of organization of files and file-systems, don't forget to check the operating systems course, C69.

6.- General Principles of Recursion

By now you should have an intuitive understanding of the following general principles that are part of every recursive problem, and will therefore form the basis of coming up with a recursive

solution to any of these problems:

* The problem itself is complex.
* But ***it can be broken down*** into smaller, or simpler versions of itself.
* The ***same process applies*** to the original problem, and to the progressively smaller/simpler instances of it.
* At some point, the solution to one of the small problems becomes easy to compute, we can stop breaking down into smaller problems and simply return the solution.
* The process now goes back building a solution for larger and more complicated subproblems until the original problem is solved.

Structure of a Recursive Solution

Because of the properties noted above, we can say that every recursive solution to a problem will have the following components:

1 - ***A Base Case***: This represents the simplest (trivial) form of the problem being solve, for which the answer can be computed and returned without additional recursion.

Examples:

- For strings, the base case *often* is an empty string, or a string with one char
- For numeric arrays, the base case *often* is an array with 1 entry, or an empty array
- For graph problems, the base case *often* involves a graph with a single node, or a graph where the node being processed has a specific property (e.g. for mapping applications, it could be that the node being processed represents the location we're looking for)
- For *search* (on lists, trees, or other data structures) the base case is *often* having reached an empty sub-list, or sub-tree, or having found the item we're searching for.

Note: As you can see there can be several different base cases for a given problem. You have to think about all that apply to your problem – we say '*often'* in the examples above because depending on the problem the base case *may be something different.* Once again you have to figure out what is appropriate given your problem. ***What is critical is to ensure there is a base case, otherwise your recursion will not end!***

2 - ***The Recursive Case:*** This is the general case, where we ***split***, ***simplify***, or otherwise ***make the problem smaller*** and ***closer to the base case***. It involves thinking about how to break down a given problem, and involves ***recursively calling*** the function with the smaller sub-problem(s). The recursive case implementation is also responsible for re-constructing the solution for the original problem from solutions for the smaller, simpler sub-problems.

Examples:

- For strings, the recursive step *often* breaks the string into chunks. Some problems will require taking out a particular character (e.g. the first, or the last one), others

will split the string in chunks at a specific point. Either way, the resulting substrings are closer to the base case.

- For numeric arrays, the recursive step *often splits* the array into a pair of sub-arrays, some problems split the first or last entry from the rest of the array, others (like *mergesort*) split the array in two. Either way the resulting sub-arrays are closer to the base case.
- For *graphs*, the recursive case will *often* perform processing on subsets of the graph (e.g. in the mapping example – a node will *ask their neighbour what the path to the desired location is*). Another possibility is that the recursive step may *split* the graph into *sub-graphs* and process each of them recursively (similarly to how *mergesort* processes an array). Examples of this would be *tree traversals*. The traversal process splits a tree into *left* and *right* subsets, and processes these recursively until reaching the base case of an empty subtree.
- For *search*, the recursive step *often* calls for recursively searching over a subset of the data structure where we expect the information we want to be. For example, for ***BST***s, the search process determines which sub-tree the desired item should be in, and recursively searches that subtree. At each step, the remaining sub-tree is closer to the base case (either finding the item we want, or reaching an empty subtree).

How to design a recursive solution:

Carefully consider your problem. ***Write down and illustrate*** an example – e.g. if you are working on arrays, draw a *representative* sample array with data.

Then consider: ***What should the the base case for this problem?***
How many different base cases are there?
Have you found all of them?

Once you know the base case(s): ***Look at your example, and ask yourself how you could split your problem into sub-problems that will get closer to the base case, and whose solution can be used to build the solution for the original problem.*** - this process will give you ***the recursive case.***

Important note: There may be several different ways to split a problem for the ***recursive case***. Choosing an appropriate way to split your problem will make the ***recursive case*** simpler and more intuitive to implement – and will likely reach the base case in fewer steps. So, ***consider possible ways to split the problem and determine which is the better one!***

This requires a bit of thought, having a written example often helps by allowing you to ***visualize different ways to split your problem*** until you find the one that works.

Once you have the ***base case*** and the ***recursive case***, check that the recursive case is able to reach a base case ***in every case***. Assuming it does, you can go ahead and implement your solution!

Example:

Let's do a complete example of how we take a problem, and come up with a good recursive solution for it by carefully thinking of what the ***base case*** and ***recursive case*** should be.

Problem: Sorting an array of integers

9	6	3	7	0	2	8	1	4	5

This is a very short array, but you know that you should have in the back of your mind the consideration of the ***complexity of your solution*** for very large arrays.

What is the base case?

The base case for sorting arrays is either an empty array, or an array with one value. Both of these cases ***are already in sorted order***. You can easily come up with examples that show that a 2-entry array ***is not in sorted order***.

How about the recursive case?

As noted above, there are many ways in which we could split our problem into sub-problems that are closer to the base case. We also said that depending on the choice we will end up with simpler/more intuitive solutions, or with more complex solutions. And that the different possibilities may result in algorithms that are less or more efficient.

Let's have a look at three different ways we could split the problem of sorting an array, each of which leads to a different sorting method. We will spend a moment thinking about the complexity of our sorting process for each of the proposed ***recursive cases***, and this will help you understand and remember that you have to think very carefully about what's the appropriate way to split the problem you need to solve.

Recursive case 1)

Sort the array by splitting the input into two sub-arrays such that

a) The first entry goes into one sub-array, and all remaining entries in the second sub-array.
b) We then ***recursively sort*** each sub-array using ***recursive case 1)*** until the base case is reached.

When sub-arrays are sorted, we ***merge*** them to build larger sorted arrays.

This is, in effect, *a variation of mergesort*. The difference is that the splitting of the array does not give two components of equal size at each step. It will sort the array, the ***recursive case*** will reach

the base case in every instance (can you show this?), and the whole process results in a correctly sorted array. ***Are we done?*** Well… let's consider the complexity of our proposed solution.

In the sample array we show above, the process works as follows:

[9 | 6 | 3| 7 | 0 | 2 | 8 | 1 | 4 | 5]	←	Input array
[9] [6 | 3 | 7 | 0 |2 | 8 | 1 | 4 | 5]	←	First application of ***recursive case 1)***
[9] [6] [3 | 7 | 0 |2 | 8 | 1 | 4 | 5]	←	Second application of ***recursive case 1)***
[9] [6] [3] [7 | 0 |2 | 8 | 1 | 4 | 5]	←	Third application of ***recursive case 1)***
.		
.		
.		
[9] [6] [3] [7] [0] [2] [8] [1] [4] [5]	←	10th application of ***recursive case 1)***
[9] [6] [3] [7] [0] [2] [8] [1] [4 | 5]	←	Merge {4} and {5}
[9] [6] [3] [7] [0] [2] [8] [1 | 4 | 5]	←	Merge {1} and {4,5}
[9] [6] [3] [7] [0] [2] [1 | 4 | 5 | 8]	←	Merge {8} and {1,4,5}
.		
.		
.		
[0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9]	←	Merge {9} and {0,1,2,3,4,5,6,7,8}

How much work does this take for an array of length N?

- We need ***N*** recursive splits to reach the ***bottom of the recursion*** (the last call time we split the problem into smaller components) ***no element swapping or comparison is needed here*** so the contribution to the algorithm's complexity is negligible.
- We then need to merge sub-arrays – For each of ***N*** levels of splitting, we need to put a total of ***N*** elements back into larger, sorted sub-arrays. At each level the total merging cost is ***O(N)***, and we have to do this ***N*** times, so the overall complexity of our sorting is $O(N^2)$. Not too good… we just re-discovered insertion sort!

Surely we can do better!

Recursive Case 2)

Sort the input array by splitting the input into two sub-arrays so that:

a) One of the array entries, called ***pivot*** is chosen at ***random***
b) Two sub arrays are created. The first one will have all the (still un-sorted) entries *less than or equal to the* ***pivot***. The remaining (un-sorted) entries go to the second sub-array.
c) ***Recursively sort*** the sub-arrays using ***recursive case 2)*** until the base case is reached

Once all sub-arrays are sorted, simply put them back together to form the entire sorted array.

This is one version of *quicksort*, where the choice of pivot is *random*. Let's see how it works in our sample array above:

9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5	← Input array

6 | 3 | 0 | 2 | 1 | 4 | 5	9 | 7 | 8	← First application of ***recursive case 2)*** pivot is ***6***
sub 1.1	*sub 1.2*	(sub arrays labeled so we can see what's going on)

recursively sort *sub 1.1* and *sub 1.2* – working on sub *1.1*

0 | 2 | 1	3 | 6 | 4 | 5	← Second application of ***recursive case 2)*** pivot is ***2***
sub 1.1.1	*sub 1.1.2*	

recursively sort *sub 1.1.1* and *sub 1.1.2* – working on *sub 1.1.1*

0 | 1	2	← Third application of ***recursive case 2)*** pivot is ***1***
sub 1.1.1.1	*sub 1.1.1.2*	

recursively sort *sub 1.1.1.1* and *sub 1.1.1.2* – working on *sub 1.1.1.1*

0	1	← Fourth application of ***recursive case 2)*** pivot is ***1***
sub 1.1.1.1.1	*sub 1.1.1.1.2*	

recursively sort *sub 1.1.1.1.1:* ***base case,*** and *sub 1.1.1.1.2:* ***base case*** *- merge*

0 | 1	2	← Back to *sub 1.1.1.2* which needs sorting
sorted 1.1.1.1	*sub 1.1.1.2*	

sub 1.1.1.2 is ***the base case****, merge*

0 | 1 | 2	3 | 6 | 4 | 5	← Back to *sub 1.1.2* which needs sorting ...
sorted 1.1.1	*sub 1.1.2*	

The process continues, taking each sub-array when its turn comes, splitting it according to ***recursive case 2)*** until the base case is reached, and merging the sorted sub arrays into larger sorted arrays until eventually the entire array is sorted

0 | 1 | 2 | 3 | 4 | 5 | 6	7 | 8 | 9	← Just finished sorting *sub 1.2* - *merge*
sorted 1.1	*sorted 1.2*	

0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9	← Done!

As we discussed in the previous unit, *quicksort* has an ***average complexity of O(N Log(N))***. If we're not unlucky with our choice of pivots, each split creates sub arrays that are roughly of equal size. In that case, we can expect to need approximately $Log_2(N)$ splits for the recursion to ***bottom out*** with all sub-arrays at the base case. For each ***level of splitting***, we have to put a total of ***N*** elements into their corresponding sub array, so the cost of the splitting step is ***O(N Log (N))***. The merging step then has to put sorted sub arrays back together – There are on average ***Log(N)*** levels, and at each level, a total of ***N*** elements must be put into bigger sorted arrays, this takes ***O(N)*** time per level. Therefore, the complexity of the merging step is on average ***O(N Log(N))***. And the total complexity of *quicksort* is ***O(N Log(N)) on average***.

We also saw that if we're unlucky with out choice of *pivot*, the splitting process puts every entry except for the pivot on one sub-array, and just the pivot in the other. If this happens at every step, the process will be identical to what we got from ***recursive case 1)*** and we're looking at a complexity of $O(N^2)$. Insertion sort again!

So ***recursive case 2)*** is much better than ***recursive case 1) ON AVERAGE***. But we still need to worry about the potential for slowdowns with an unlucky input array.

Recursive case 3)

Sort the input array by splitting the input into two sub-arrays such that:

a) Each sub-array has half the entries of the original (if the input has an uneven number of entries, one sub-array will have one entry more than the other).
b) Recursively sort these sub arrays using ***recursive case 3)*** until the base case is reached.

Sorted sub-arrays are *merged* to build larger sorted arrays until the original input is sorted.

This is the *mergesort* algorithm we have discussed in this Unit. Let's see how it works on our sample array:

9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5	← Input array

9 | 6 | 3 | 7 | 0	2 | 8 | 1 | 4 | 5	← First application of ***recursive case 3)***
sub 1.1	*sub 1.2*	(sub arrays labeled so we can see what's going on)

recursively sort *sub 1.1* and *sub 1.2* – working on sub *1.1*

9 | 6 | 3	7 | 0	← Second application of ***recursive case 3)***
sub 1.1.1	*sub 1.1.2*	

recursively sort *sub 1.1.1* and *sub 1.1.2* – working on *sub 1.1.1*

9 | 6	3	← Third application of ***recursive case 3)***
sub 1.1.1.1	*sub 1.1.1.2*	

recursively sort *sub 1.1.1.1* and *sub 1.1.1.2* – working on *sub 1.1.1.1*

9	6	← Fourth application of ***recursive case 3)***
sub 1.1.1.1.1	*sub 1.1.1.1.2*	

recursively sort *sub 1.1.1.1.1:* ***base case,*** and *sub 1.1.1.1.2:* ***base case*** *- merge*

6 | 9	3	← Back to *sub 1.1.1.2* which needs sorting
sorted 1.1.1.1	*sub 1.1.1.2*	

sub 1.1.1.2 is ***the base case****, merge*

3 | 6 | 9	7 | 0	← Back to *sub 1.1.2* which needs sorting ...
sorted 1.1.1	*sub 1.1.2*	

The process continues, taking each sub-array when its turn comes, splitting it according to ***recursive case 3)*** until the base case is reached, and merging the sorted sub arrays into larger sorted arrays until eventually the entire array is sorted

0 | 3 | 6 | 7 | 9	1 | 2 | 4 | 5 | 8	← Just finished sorting *sub 1.2* - *merge*
sorted 1.1	*sorted 1.2*	

0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9	← Done!

As we discussed earlier in this Unit, *mergesort* has a ***guaranteed worst case complexity of O(N Log(N))***. So in terms of ***sorting performance***, ***recursive case 3*** gives us the optimal solution!

Notice the following:

- ***recursive case 1)***, and ***recursive case 3)*** both have very simple splitting procedures.
- ***recursive case 3)*** easily beats ***recursive case 1)*** in terms of ***sorting complexity***.

- ***recursive case 2)*** has a more complex splitting process, and its ***complexity*** is in between the other two, depending on the particular input provided to the algorithm.

Conclusion: As you can see, the choice of how to split a problem ***matters greatly***. Depending on your choice, you may obtain a simpler, more intuitive algorithm, and you may have a significant difference in complexity. So, it's important that you always spend a good amount of time and effort into thinking through the different possible recursive cases for your problem, and then choose the better one.

Friendly reminder: Do not forget what we discussed in Unit 4. Differences in implementation for different algorithms matter, in the case above, we concluded that *mergesort* is better than *quicksort* based on the fact that it has a *guaranteed complexity of* ***O(N Log(N))*** whereas *quicksort* can be slow if we're unlucky. But remember we tested *quicksort* on a task of sorting multiple arrays with randomly ordered entries, with array lengths in the hundreds-of-thousands, and we *didn't find it slowed down*.

Indeed, the documentation for the library implementations of common sorting methods including *quicksort* and *mergesort* says the following:

```
Normally, qsort() is faster than mergesort() is faster than heapsort().
Memory availability and pre-existing order in the data can make this
untrue.
```

Exercise: Implement *mergesort* from our description and examples above. *Test* it against the standard implementation of *qsort()* (quicksort) from the standard C library, for large input arrays. See how it compares.

7.- Implementation considerations for recursive methods

Recursion requires that a function ***call itself*** a (possibly large) number of times. Remember from all the time we spent looking at the memory model in Unit 2, that every time we call a function, ***space has to be reserved for the function's parameters, variables, and return value***. In non-recursive code, we don't usually worry about it because we know that after a function is called, its work completed, and a return value obtained, all the space reserved for the function is released.

However, with recursive code this is tricky – all the space reserved ***will eventually be released***, but not until each recursive call is completed. What that means is ***space reserved for function calls early in the recursive process will hang around until it receives results from the recursive calls it launched.***

To understand this problem, let's have a look at a short example of a function that sums up an array of integers recursively (as we've noted before, this is not a problem where we would immediately think of recursion, but it's simple enough it will help us illustrate how recursion is working!).

The function that sums the array works as follows (pseudocode):

```
sum(array, n)                                          : L1
     if (n==0) return array[0];                        : L2
     else return array[n] + sum(array, n-1)            : L3
```

The function takes an input array, and recursively computes the sum of its elements. The ***base case*** is an array with a single entry in which case the sum is simply the value for that element (if we're careful, we also need to add a case where the array is empty, in which case the sum is zero).

The input parameter ***n*** is used to indicate the part of the array the function is adding – it starts at the index of the last element in the array, and with each recursively call, decreases by one. This ensures that we get closer to the ***base case***. Once the base case is reached, the recursion rebuilds the sum from the first element back to the last.

Let's see how this works:

Step 1 - The first call to the function (from somewhere in your program):

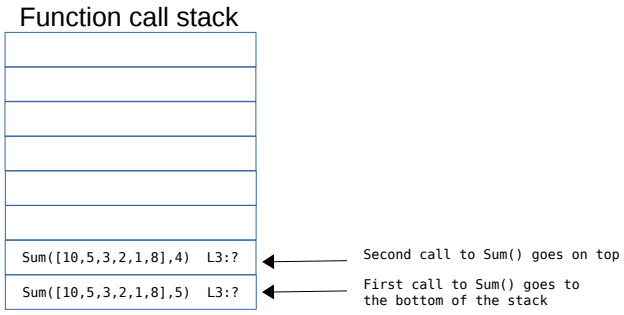
```
Sum([10,5,3,2,1,8],5)
```

here, n=5, last entry in the array (the 8), it's not the base case, so the function goes to line 3 (L3) and needs to return array[5] plus the results of a recursive call with n=4.

Step 2 – Recursive call to Sum() with n=4

```
Sum([10,5,3,2,1,8],4)
```

At this point, *there are two active calls to Sum()*, each with their own value of n. The computer needs to keep track of both of them. This is done by using a ***stack***. A ***stack*** is nothing more than an array that has the particular property that we fill it up from the end and work out way backwards – the end of the array is called the ***bottom of the stack*** and the beginning of the array is called ***the top of the stack***. In our example above, somewhere in memory we will have the following information in the stack:

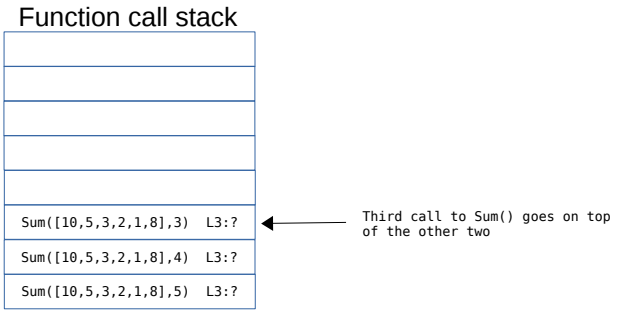


The second call to Sum() has n=4, which is not the base case, so it needs to return array[4] plus the result of a recursive call:

Step 3 – Recursive call to Sum() with n=3

```
Sum([10,5,3,2,1,8],3)
```

This adds another active call to Sum() at the top of the stack



The value of n=3 is still not the base case, so another recursive call is needed. This will continue until we reach n=0.

Step 4 – Recursive call to Sum() with n=2

```
Sum([10,5,3,2,1,8],2)
```

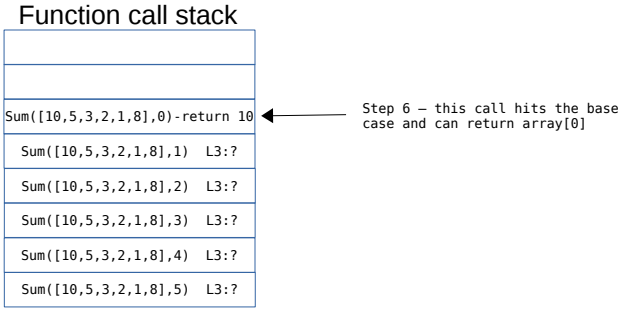
Step 5 – Recursive call to Sum() with n=1

```
Sum([10,5,3,2,1,8],1)
```

Step 6 – Recursive call to Sum() with n=0

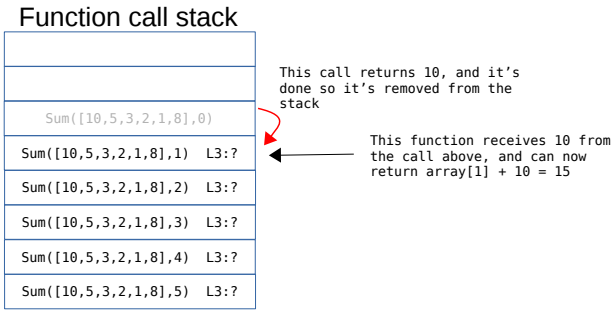
```
Sum([10,5,3,2,1,8],0)
```

At this point, the function call stack will look like this:

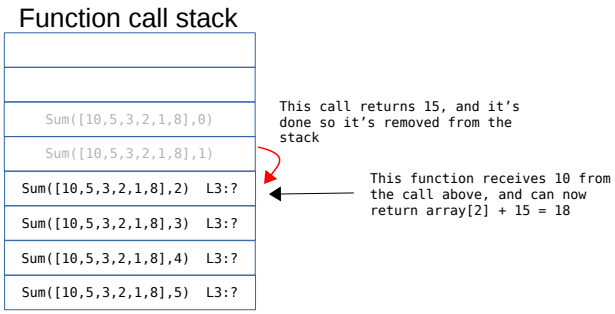


Step 7 – The call to Sum() at the ***top of the stack*** finds the base case and returns array[0] (which

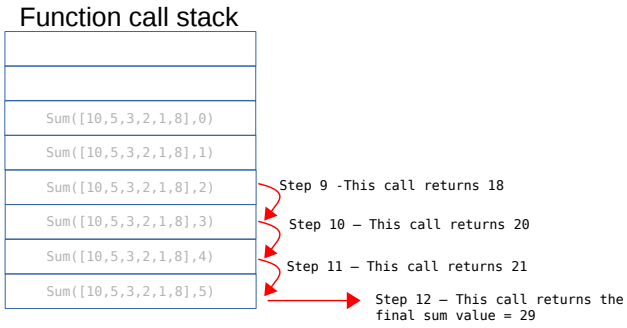
has a value of 10).



Step 8 – The next call now at the top of the stack (with n=1) now received a 10 from its recursive call, and can return array[1]+10.



The process continues with each call now completing their work and returning a value to the next one below, until the call to Sum() at the bottom completes and returns the correct sum for the whole array.



At the end of the process, all the active function calls have been completed and the stack is empty again. The entire process for reserving space in the stack for each function call is done automatically, what you need to keep in mind is that each recursive call will require its own section of the stack – this means if you have a recursion that requires a very large depth to reach the base case, or if you have a recursion where each call requires a lot of stack space, you can run intro trouble by running out of space – ***a stack overflow***.

Important concepts to remember:

- The way computers work, memory for functions in a program is stored in a ***stack***. A stack is simply an array (in this case, of memory boxes!) with a fixed size, it's filled from the ***last entry (bottom of the stack),*** to the ***first entry*** (one way to visualize it is like a bucket, when you toss something into it, it goes on top of everything else already there).
- The stack ***grows*** upward through the boxes reserved for it (as shown in the diagram above)
- The part of the stack reserved for each ***function call*** is called a ***stack frame*** (in the diagram above, each stack frame is shown as a call to the function with corresponding parameters).
- Your code can run out of stack space with recursive functions, if you're not careful to ensure the recursive process reaches the base case in a reasonable number of calls.
- If the code runs out of stack space, it will be terminated (crash, segfault, etc.)
- ***Every call to the recursive function gets its own reserved space for variables, parameters, and return type*** - therefore the data used for each recursive call is separate from all others, and there is no possibility that anything can get overwritten or changed accidentally unless you specifically provided pointers so the recursive call can change things outside its stack frame.

You should keep the above diagram in mind - it shows that if you're not careful with recursion, and if you don't write your code keeping in mind how many parameters and variables you actually need, you can quickly run into memory problems by using up all the stack space reserved for your program.

It should also make you think of the ***overhead*** caused by reserving all that memory and passing parameters and return values around during a sequence of recursive calls. ***Carelessly written recursive code can be quite inefficient*** because of all that overhead.

However, if we are smart and write our code properly, we can take advantage of a technique called ***tail-recursion, or tail-recursive optimization***.

Tail Recursion

In the example above, the '*Sum()*' function reconstructs the result we want from the partial sums returned by recursive calls – the last recursive call (the one that reaches the base case) returns a value, which then is used to compute and return the next partial sum, which then is used to compute and return the next partial sum, and so on all the way back to the first call to *'Sum()'* which computes and returns the final sum value.

This line of pseudocode takes care of this part:

```
else return array[n] + sum(array, n-1)                : L3
```

The thing to notice is that ***once the recursive call to 'sum()' returns a value, we still need to***

add array[n] before we can return from the current call. Because the function still has work to do after the recursive call is complete, we need to have access to the local variables and parameters for this function call (which, as we know, are stored in the stack).

However, ***we could achieve the same result in a slightly different way***. Instead of having the sum of the array be re-constructed backward from each successive recursive call, we could ***pass the partial sums forward into the recursion*** – so that the final recursive call (the one that hits the base case) can compute the final sum value and simply return it!

Let's have a look at how we could write the sum function in that way:

```
Sum_TR(array, n, part_sum)                                    : L1
     if (n==0) return part_sum+array[0]                       : L2
     else return Sum_TR(array,n-1,part_sum+array[n])          : L3
```

This version is very similar to our original one, but the difference in line L3 turns out to be quite significant:

```
     else return Sum_TR(array,n-1,part_sum+array[n])          : L3
```

In this version, ***the function returns the result from the recursive call directly – it doesn't need to do any further operation or compute any further value before it can return***. Put in a different way, ***the recursive call is the last thing the function does before returning a result***. We call this form of recursion ***Tail Recursion***.

Let's see how this may change things when the function is called:

Step 1: First call to Sum_TR with *n=5*, *part_sum=0*

```
Sum_TR([10,5,3,2,1,8],5,0)
```

In the stack, this will generate one stack frame for this call with *n=5*, which is not the base case, so we have a recursive function call with *n=4*, and with *part_sum = 0 + array[5] = 8*:

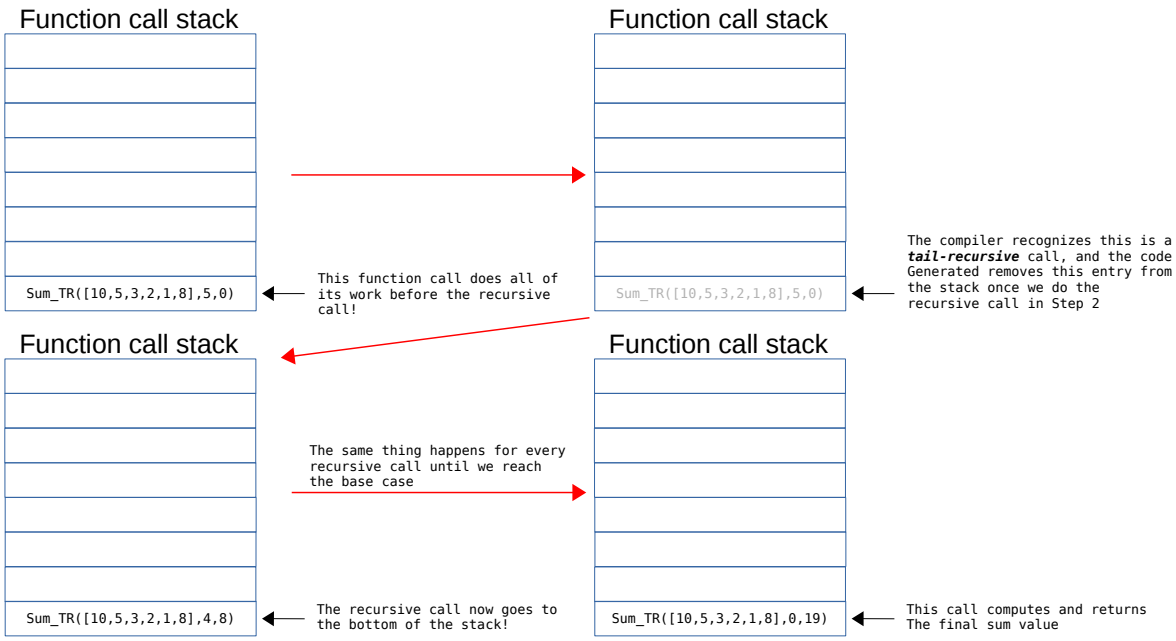
Step 2: Recursive call to Sum_TR, n=4, part_sum=8

```
Sum_TR([10,5,3,2,1,8],4,8)
```

Here's where something good happens: ***Because the call from Step 1 has already done all the work it needs to do, and passed forward a partial sum, its local variables and parameters are no longer needed and can be removed from the stack!*** The compiler is designed to recognize this, and the code generated will ***not fill up the stack with data for each recursive call***.

Instead, each call's stack frame exists only until the moment when it executes the next recursive

call, at which point that stack frame is removed. The illustration below shows how this works:



The remaining steps involve:

Step 3: Recursive call with *n=3*, *part_sum=9* (8 + array[4])

```
Sum_TR([10,5,3,2,1,8],3,9)
```

Step 4: Recursive call with *n=2*, *part_sum=11* (9 + array[3])

```
Sum_TR([10,5,3,2,1,8],2,11)
```

Step 5: Recursive call with *n=1*, *part_sum=14* (11 + array[2])

```
Sum_TR([10,5,3,2,1,8],1,14)
```

Step 6: Recursive call with *n=0*, *part_sum=19* (14 + array[1])

```
Sum_TR([10,5,3,2,1,8],0,19)
```

This call reaches the base case, and returns the final value, *array[0]+part_sum* = 29. Notice that now ***we don't have to go back up the stack re-building the solution!*** it's already complete the moment we reach the base case. ***There never was a set of active function calls in the stack***. The only active call at any time is the ***call currently being processed***.

So, by changing the way we write out recursive function a little bit – in order to pass partial

results ***forward*** rather than re-buliding from partial results ***backward*** we have achieved the following:

- Our recursive call no longer takes up large amounts of stack space to keep track of all the active function calls.
- The result is available as soon as we reach the base case

Because of these two advantages, ***tail recursive*** functions can handle ***inputs that are much larger and require much deeper recursion to reach the base case***, and ***we eliminate (or at least significantly reduce) the overhead of performing the recursive calls caused by all the stack management required by non-tail-recursive calls.***

Modern compilers are able to take advantage of recursive functions written in this way, and the resulting code is as efficient as an equivalent loop. Let's see how the three ways of implementing the sum function compare – with loops, with tail-recursive calls, and with non-tail-recursive calls – the following is the output of a comparison involving an array with 10,000 floating point numbers, and computing the sum of the array 100,000 times (so we can measure the time accurately, otherwise a single call to sum is too fast to measure on modern computers!):

```
./a.out
For loops, sum=4988.303693, time taken=0.805763
Recursion, sum=4988.303693, time taken=1.389484
Tail Recursion, sum=4988.303693, time taken=0.797839
```

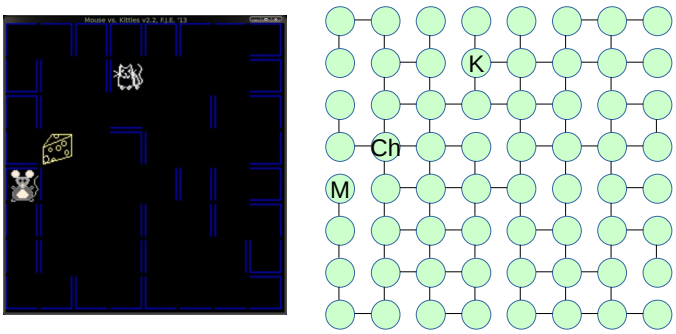
As you can see, the tail-recursive call is as fast as the function that uses a simple for loop! In terms of memory, it's equally efficient as well. The bottom line is, when writing recursive code you want to try to write it so that it is ***tail-recursive***. It pays out in terms of performance, and allows you to handle much larger problems.

Note: You can learn a lot more about recursion, tail-recursion, stack frames, interesting applications of recursion, and get a lot of practice in recursive thinking by taking the Programming Languages course, C24.

Important note: Any problem you can solve with iterations can also be solved with recursion, and vice-versa. Both of these are general tools for problem solving. The difference will be in how natural your solution will be, and how easy to implement, maintain, and extend once written. Choosing the right tool is important.

8.- Back to Graphs

We started this Unit by learning about graphs and the kinds of problems we can solve with them. We said that recursion is a natural tool for working with graphs since graphs are recursive in nature. Now that we have learned about recursion, let's go back and solve the path-finding problem (from Artificial Intelligence) that we described above.



In the problem above, we have a start location (the mouse), a destination (the cheese), and a graph that has 8x8=64 different locations. Each of these is represented by a node. We could number the nodes as follows:

```
0  1  2  3  4  5  6  7
8  9  10 …
.
.
.
.                  …  62 63
```

And we have ***an adjacency matrix*** of size 64x64 (it's too large to copy here!) that as a ***one*** at entry ***A[i][j]***, and also at entry ***A[j][i]*** if nodes ***i***, and ***j*** are connected (there is no wall between them).

The problem we have is: ***Find a path, which consists of a sequence of connected nodes, leading from the start node to the destination***.

We already discussed informally what the solution looks like (asking neighbours, who ask their neighbours, who ask their neighbours…) but at this point we should think about the recursive solution in a more concrete way, and consider how to formulate the ***base case*** and the ***recursive case*** for this problem.

Base case: Consider the following -

Unless the destination node is a neighbour, we don't know the path to the destination

With that in mind

The base case is: ***We have reached the destination node*** and we can tell its neighbours which way to go to get there.

Is that the only base case?

What if we have a graph such that there is no path from the start node to the destination node (e.g. the cheese and the mouse are separated by walls with no way around)? *In that*

case we can expect at some point to have asked for directions all nodes that can be reached from the start node, and we didn't reach the destination.

So another base case is: ***There are no nodes left to explore in looking for the destination.***

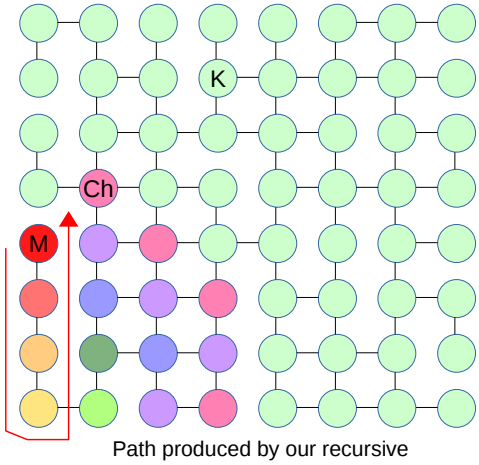
Recursive case: The recursive case deals with finding a path from a given node (let's call it the *current* node) to the destination. Because of the requirement that the path consist of a sequence of connected nodes, the recursive case consists of:

- Removing the *current* node from the graph (to create a smaller sub-graph)
- Recursively finding a path ***from each neighbour of the current node to the destination***

Once a path has been found from ***any*** of the neighbours, the path from the *current* node is obtained by attaching the *current* node to the start of the path.

Because the recursive case removes one node from the graph with each recursive call, we are sure that it will eventually reach one of the two base cases above.

This process will result in the path illustrated earlier:



Path produced by our recursive solution. Neighbours explored at each recursion level have the same colour

The ***order*** in which we check for a path from the neighbours to the destination is important. Different orderings will result in different search patterns, different nodes being explored, and possibly also a different path being found. You can learn all about the different exploration strategies (including optimizations to reduce the computational complexity of the process!) in the AI course D84. For now, let's look at the pseudo-code of the simplest strategy – one that is purely recursive and doesn't require you to keep additional data structures to keep track of which neighbours remain to be processed for each recursive call to the procedure. This exploration strategy is called ***DFS*** for ***Depth-First Search*** and it is a fundamental technique in graph search. ***DFS*** forms the basis of algorithms used for scheduling and other constrained search problems.

```
// Inputs: current node (its index in the graph)
// destination node (its index in the graph)
// Adjacency matrix (to figure out a node's neighbours)

DFS(current, destination, A)

     if current==destination
          return destination   // We got to base case!

     else
          for each neighbour of current
               subpath=DFS(neighbour, destination, A)
               if  subpath found
                     // prepend current to subpath to build full path
                     return  {current → subpath}

          return (nil)   // Second base case, no subpath from current
```

Even though we were working on a toy problem, the path finding process we discussed above applies to the general case of finding a path from one node to another through any graph. As such, you could apply DFS to a grid representing a city map, a graph for a computer network, or any other path finding problem you can think of!

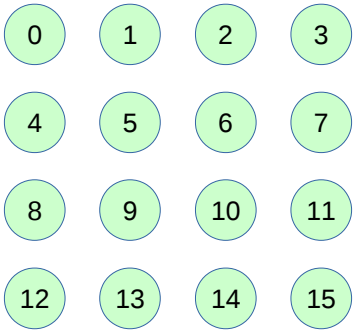
Exercise (CRUNCHY!): Implement DFS on a grid such as the one shown above. ***You don't need to actually create data for nodes in the graph!*** Since we are not actually doing any information processing at this point (only finding paths from one node to another by their index in the grid) it is enough to create an ***adjacency matrix*** (for a grid of size ***m x n*** the adjacency matrix has a size of ***mn x mn***).

Here's a sample adjacency matrix you can use:

```
int Adj[16][16]={{0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
                 {1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                 {0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
                 {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0},
                 {0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
                 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0};
```

The above matrix is for a ***grid*** of size *4x4*. As we discussed, an entry ***Adj[i][j]*** is one if the node with index ***i*** is connected to the node with index ***j***. So for example, in the above matrix, ***Adj[5][1]=1*** so nodes 5 and 1 are connected.

Complete the diagram below by adding edges between nodes that are connected, according to the adjacency matrix above.



Now that you know what the grid looks like, implement your DFS code. Test it by finding paths between different pairs of nodes in the graph and checking that they make sense for the grid we have (e.g. the path nodes should sequentially go from *start* to *destination* and each pair of nodes is connected)

Note: The DFS process includes a step where the *path* is grown by prepending the *current* node to a *subpath* returned by a recursive call to DFS. ***This step requires thought***. You're expanding a data structure whose size is not known in advance (you don't know the length of the path you will find). So think carefully about how to implement this, and remember we have several data structures we can use at this point which allow you to add data on-demand.

9.- Debugging Recursive Code

Testing and debugging recursive code can be tricky. But with a bit of practice, and if you think carefully about what your recursion is supposed to be doing, you'll be able to handle any recursive function.

What you need to understand before you start debugging

- Your base case (review that you have every base case covered, and that they are implemented)
- Your recursive case (review that the splitting is reasonable, and implemented properly)

Understand how the recursive process is supposed to work, so you can determine if it is doing the right thing once you start tracing.

Tracing your recursive code

- The tricky part here is that the same function gets called over-and-over-and-over. So you need additional information in order to figure out ***which*** of the many recursive calls is not doing the right thing, and at ***what point*** things go wrong.

How can you do this?

a) Expand your recursive function call with an integer variable called 'level' or 'depth', which is increased by one each time you call the function. For example, with the DFS function above, we can modify the DFS call to look like this:

DFS(current, destination, Adj, depth)

The first call to DFS will look like this:

```
DFS(15, 1, Adj, 0);
```

(Find a path from node 15 to node 1, with adjacency matrix Adj, initial recursion depth is 0).

Within the DFS code, the recursive calls will look like this:

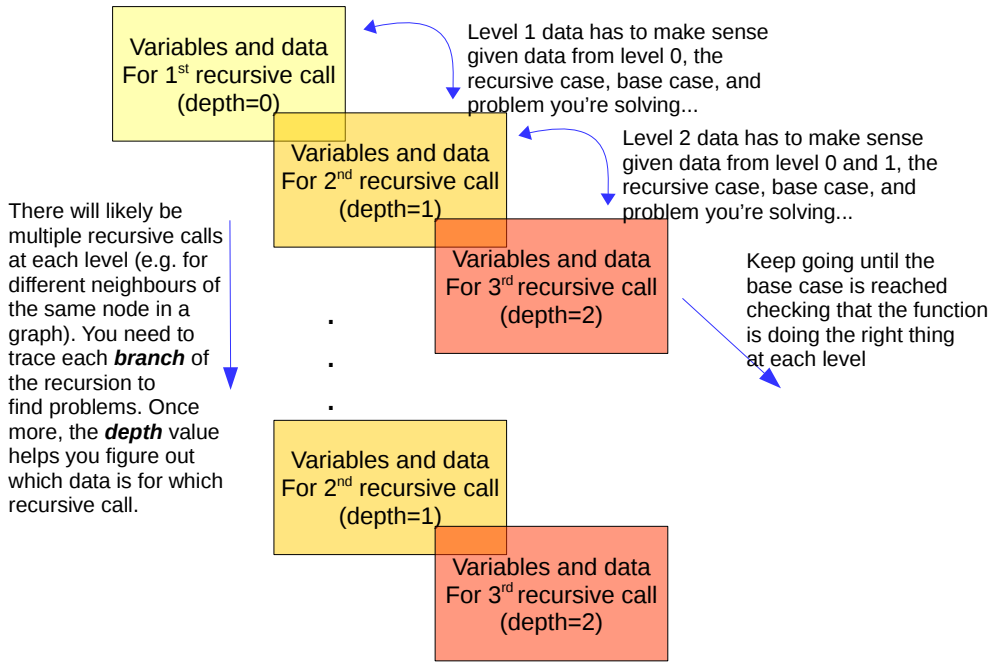
```
subpath=DFS(neighbour, destination, Adj, depth+1);
```

so in effect, every time we recursively call DFS, the depth is increased by one.

How does this help?

When inspecting the value of variables and parameters within a recursive call, we need to know at what level in the recursion process we are in. The *'depth'* variable tells us this information. So now, we can either use a debug tool (such as gdb), or we can *printf()* the variables and parameters we need in order to debug our code - while also being able to tell *which recursion level these variables are from.*

Look at the whole picture:



Recursion creates a stack of function calls, so you need to print variables and parameters for each of them, and ***make sure that the data in each level makes sense and is correct given the level before***. This is the part where you have to use your understanding of the recursive process, the base case, the recursive case, and the problem itself to figure out ***which part is broken, and how***.

But you have all the information you need to do this! If this looks challenging at first, remember: ***like everything else we've been doing in this course, it's just a skill you can and will master with practice.*** So, ***practice!***

Common problems with recursion:

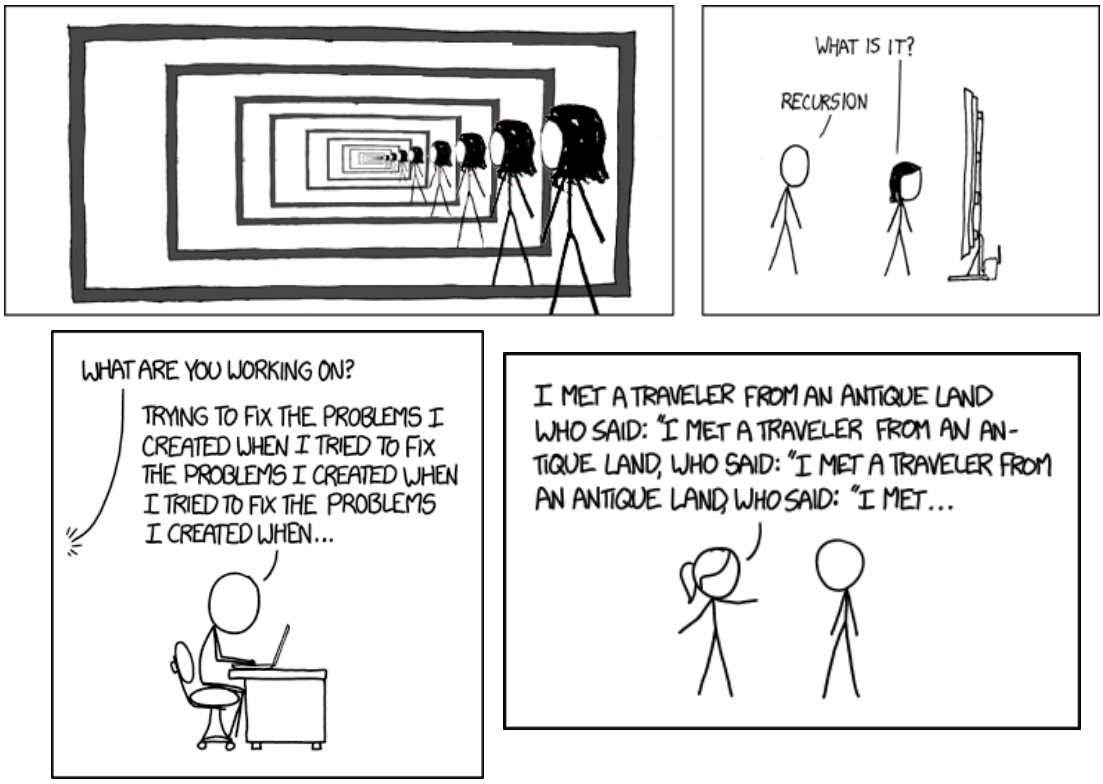
- It never reaches the base case (you can see the depth variable keeps increasing to unreasonable values) - check your recursive case, make sure it's making the problem smaller, and then check that the base case(s) have all been identified and implemented.

- It runs out of stack space for recursive calls: It can happen with large problems which require very deep sequences of recursive calls. You can use the ***'depth'*** variable to enforce a ***maximum allowed recursion depth***. Your recursive call checks the depth value, and if it is equal to the ***maximum allowed recursion depth***, it prints a message to let the user know the problem is too complex to solve with the available stack memory, and returns (it should return the equivalent to not having found a solution).
  The advantage of doing this is that it prevents your code from crashing

- The recursion doesn't return the correct solution - check that your base case is returning a reasonable result, and then check the process of building a bigger solution from the smaller one.

Note: Once you're satisfied that your solution works, you should remove the *'depth'* variable from the recursive function definition (unless you want to use it to enforce a ***maximum recursion depth*** limit).

That's all for recursion and graphs... for now! We can promise you that both of these will show up in many places and contexts in the near and not-so-near future as you advance through your career. So don't forget what you learned here! For now, here's a few cartoons that make recursive sense!



Drawing 7: XKCD cartoons about recursion. Courtesy of Randall Munroe, xkcd.com

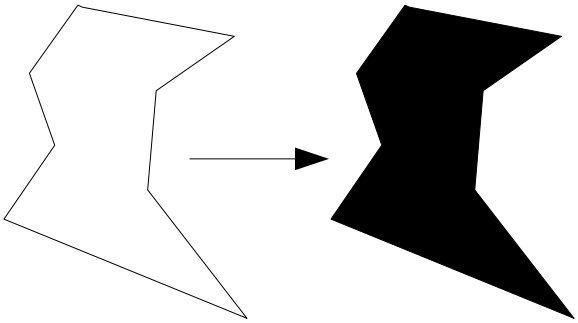
Additional Exercises and Problems

The importance of choosing the right tool - We have claimed that you can implement any algorithm in either a recursive way or with loops. Choosing the right tool for a job matters. See if you can figure out how to solve the following problems using the ***tool indicated with each exercise.***

Ex0 - Assume we have a ***BST*** that stores ***unique integer keys***. Write the algorithm that performs ***in-order BST traversal*** on this ***BST*** and prints the ***keys*** (it should print the keys in sorted order). However,

you ***are not allowed to use recursion***. Use only loops to solve this task. You can use helper data structures as needed (hint, somehow you have to keep track of the information the recursion keeps for you).

Ex1 - Flood-fill - a common operation in paint programs is that of ***filling a region with colour***. This is called flood-fill because we can think of it as pouring paint into the region, and that paint spreads out until it reaches the boundary of the region. Let' see how we would do that in a simple black & white image.



Drawing 8: Flood-filling a shape - we are supposed to fill with black the inside of the shape defined by the black border

For out purposes here, the image is a 2D array of size (500x500), black pixels have a value of 0, white pixels have a value of 1.

a) ***Write*** the algorithm for performing ***flood fill*** on figures such as the one above ***using only loops***.

b) ***Write*** the algorithm for performing ***flood fill*** on figures such as the one above ***using recursion***.

Ex3 - Graph distance - One common problem in graph-based applications is finding what subset of nodes is reachable from a given graph node within a certain number of steps. For example, suppose we have a graph representing the pre-requisite structure of courses at UofT, and we want to know which courses have CSCA48 as a pre-requisite, or as a pre-requisite to their pre-requisite. This would require us to find any nodes that are up to ***two hops*** away from CSCA48 in the graph (one hope means moving from one node to the next via a connecting edge).

Given the ***adjacency matrix*** for the graph:

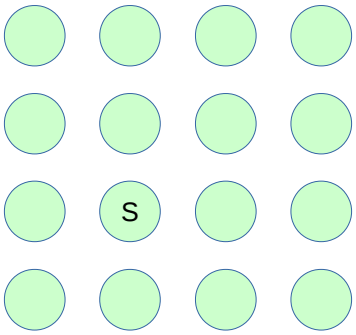
a) Propose an algorithm that determines all nodes within distance 2 of a selected start node ***using only loops.***

b) Propose an algorithm that determines all nodes within distance 2 of a selected start node ***using recursion***.

Ex4 - In the text above, we studied ***depth-first search*** for path-finding in graphs. The order in which

the neighbours of a node are expanded is relevant. In the graph below, ***label each node in the order in which it would be explored by DFS*** assuming that for each node processed by ***DFS***, the neighbours are processed recursively in the order ***top, right, bottom, left***.

The first node (the one we originally call DFS for) is marked with an 'S' (for 'start').



Be careful - you need to think about how the recursive calls to DFS will be ordered!

Ex5 - (Crunchy!) - DFS is only one possible way to explore a graph. A different method, called ***breadth-first search*** uses a different order of exploration. For ***DFS***, a node's grand-children, and great-grand-children, and so on, will be explored ***before*** all of the node's children are explored. With ***BFS***, the node's children ***are all explored before any grand-children***. Then ***all of the grand-children are explored before any great-grand-children***, and so on until the entire graph is explored.

BFS is normally implemented using a ***queue*** (no need for recursion!). In the space below, try to come up with the algorithm that explores the graph in ***BFS*** order, assuming you have access to a ***queue data structure*** - it lets you add items to it (always at the end), and lets you remove items from it (always from the front). If you want to review how queues work, see the related exercise in Unit 3!

Ex6 – Tail Recursion – Implement the Sum() function discussed in the text using for loops, using non-tail recursive calls, and using tail-recursive calls. Then compare its performance on arrays of different size (and see how large an array the plain recursive solution can handle before the stack overflows).

Ex7 – Tail Recursion – Now that you got practice with tail recursion from *Ex6*, see if you can figure out how to implement the classic examples of recursion – computing the ***factorial*** of a number, and obtaining the ***n^{th} Fibonacci*** number, using ***tail recursion***.

Note: Once more, remember that these two problems are examples of tasks that you would want to implement with ***loops***! They are here ***only to help you develop the skill of writing tail recursive code!***