Chapter 5 Graphs and Recursion

At 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 Chapter, 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.

5.1 Graphs

In computer science, **graphs** are used as a model to represent items of interest, where the **relationship between items is relevant** to the problem we wish to study or solve. To understand this, let's take a look at the sort 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 in previous Chapters.

Thus far, we can work with (possibly very large) collections of data items, these items can be regular **C** types, or **CDTs** 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** from the rest.

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 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 certain kinds of food more than others. The tools we have available up to this point do not allow us to study any of these meaningful connections.

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 Scorsese?*, and *What*

is the fastest route to get from my home to the wonderful pizza place that I love best?

Graphs store **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 because of that, they allow us to **find meaningful interactions** between data items, and to **look for interesting patterns** in complex data sets. Because of this, **graphs** are central to a large variety of methods in machine learning, artificial intelligence, and data visualization. They are used to represent information that can be conceptualized in the form of a **network** - that is, **nodes** and their **connections**. We already know what a **node** is in terms of data representation and storage, let us look at the **connections** and what they may mean. Typical applications of **graphs** include:

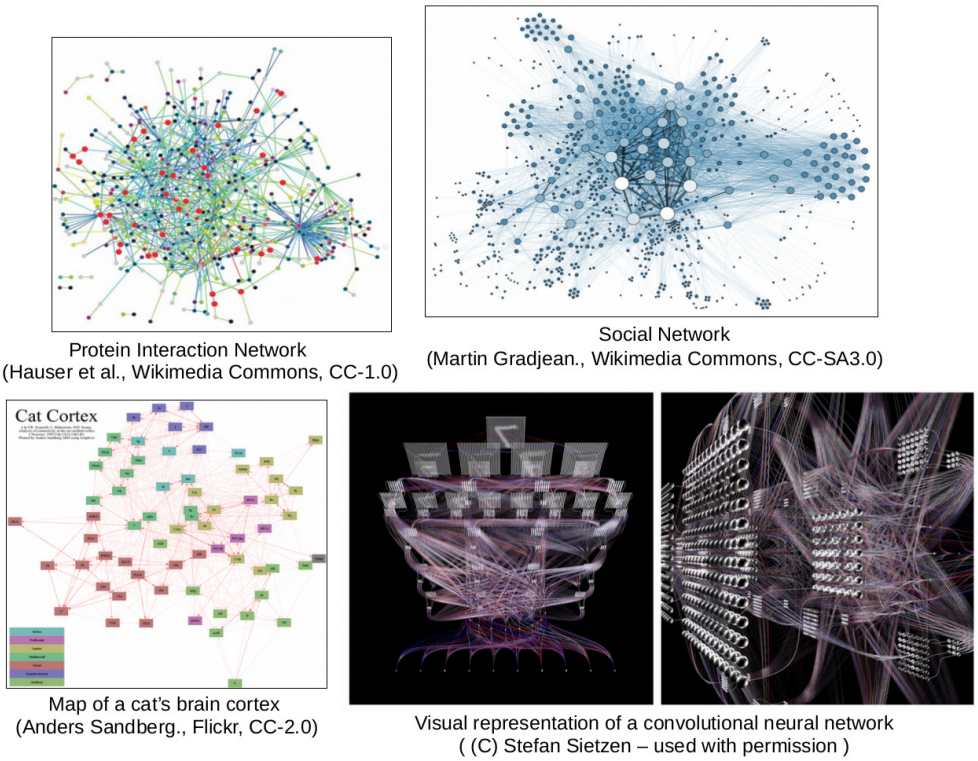
- Social networks - where **nodes** represent people, and **connections** join pairs of people who interact socially in some particular context.
- Transportation - where **nodes** represent locations, and **connections** join pairs of locations that we can travel between (for example, two intersections joined by a street). This also applies to **internet routing**, **electrical or water distribution** grids, and other similar problems.
- Genomics and bio informatics - where **nodes** can represent things like proteins, DNA sequences, enzymes, etc., and **connections** join together pairs of these that interact within a biological system we are modelling.
- Courses and their pre-requisites - where **nodes** represent courses, and **connections** represent the pre-requisite structure (if there is one) for pairs of courses to be taken.
- Databases - where **nodes** represent data tables (we briefly read about them in the previous Chapter), and **connections** represent the structure of the data in the Database. For example, one table may contain information about a company's employees, a separate table may contain information about payroll, and the connection between them indicates that the payroll table provides information about the salaries of employees in the employee table.
- Recommendation systems, which suggest items of interest based on a user's tastes - **nodes** represent items of interest (e.g. movies, books, music, or products in an online store), and **connections** represent **similarity** so that the system can decide which items to recommend for a particular user.

The above is just a small sample of the wide range of applications for **graphs**. Because they can be conceptualized as **networks**, and because they represent **relationships** between data items, **graphs** can often be **visualized** by plotting **nodes** in some particular pattern, and showing the **connections** between them - when designed carefully, such visualizations can help humans get an overview of the data they are working with at a glance, and to visually discover interesting patterns that may be worth exploring with software. Examples of graph visualizations are shown in Fig. 5.1.

5.1.1 Definition of a graph

A **graph** consists of:

- A set V of **nodes** corresponding to data items we are working with. In books, lecture notes, and future courses, you may find the term **vertex** is used instead of **node**, they are the same thing. The size of V (the number of **nodes** in the **graph**) is N.



Protein Interaction Network
(Hauser et al., Wikimedia Commons, CC-1.0)

Social Network
(Martin Gradjean., Wikimedia Commons, CC-SA3.0)

Map of a cat's brain cortex
(Anders Sandberg., Flickr, CC-2.0)

Visual representation of a convolutional neural network
((C) Stefan Sietzen – used with permission)

Figure 5.1: Sample visualizations of **graphs** in different domains.

- A set E of **edges** which are the connections between nodes. They represent the relationships existing between data items in our collection. The size of E (the number of **edges** in the **graph**) is M.

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

We have already been working with **graphs**! Trees, such as **BSTs** are **graphs** (the 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.

5.1.2 Types of Graphs

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

- **Un-directed graphs**: Edges between nodes are shared, and the relationship between the nodes goes both ways. An example of an **un-directed** graph is shown in Fig. 5.2.
- **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 **BSTs** you've been working with: Edges in **BSTs** 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 in Fig. 5.3.

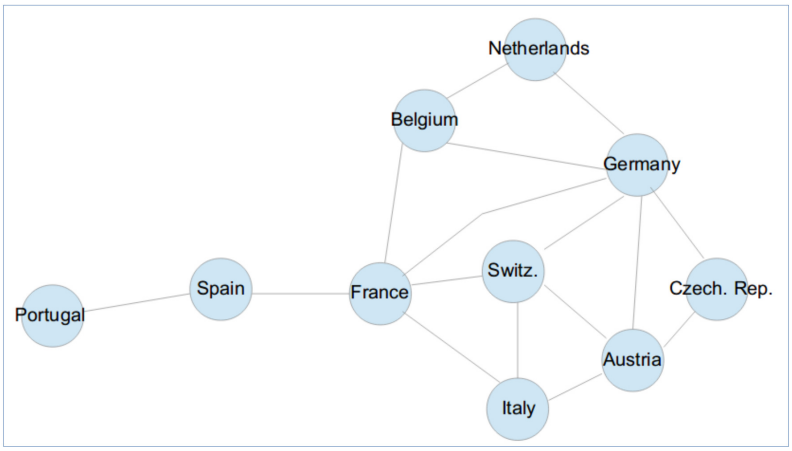


Figure 5.2: **Un-directed graph** representing countries in Europe. Edges in this graph indicate the corresponding countries have a shared land border (this relationship goes both ways by definition).

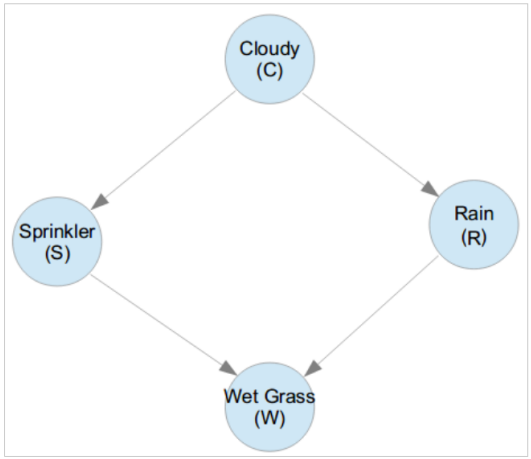


Figure 5.3: **Directed graph** representing binary (TRUE/FALSE) variables in an **inference problem**. The direction of the edges (represented by arrows) indicates which variables have a direct effect on each other. For instance, the **Rain** variable (indicating whether or not it has been raining) can directly affect the value of the **Wet Grass** variable. The converse is not true - wet grass can not direcly cause rain to happen.

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

5.1.3 Terminology for graphs

The following are definitions of important terms related to **graphs** that are often used in studying graph-based algorithms:

- **Neighbours:** For **un-directed graphs**, a **node v** is a neighbour of **node u** if there exists an edge joining both nodes - we say the two nodes are **adjacent**. For **directed graphs**, a **node v** is an **out-neighbour** of **node u** if there is an edge **from u to v**. Conversely, **v** is an **in-neighbour** of **u** if there is an edge **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-neighbourhood**, corresponding to the sets of **out-neighbours** and **in-neighbours** 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**.
- **Path:** A **path** through a **graph** is a sequence of consecutive nodes that can be visited by following existing edges between pairs of connected nodes. Figure 5.4 shows an example: 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!)
- **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.

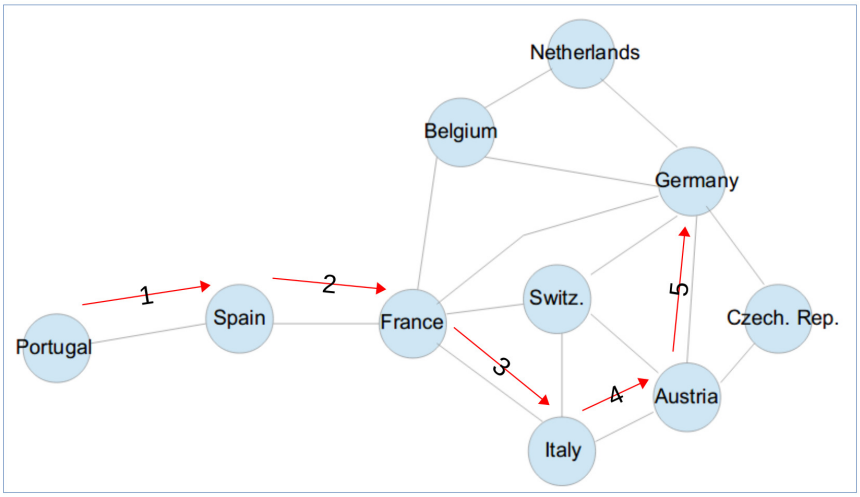


Figure 5.4: Example of a **path** in a **un-directed** graph. The **path** is the sequence of nodes visited while going from one **node** to **another**.

Note

For **un-directed graphs**, we can travel along edges in either direction while forming a path. But for **directed graphs**, we can only go from **node u** to **node v** if an edge exists that goes from **u** to **v**. This is very much like travelling in a city with lots of one-way streets, we must follow the direction of the streets while going from one place to another.

✍ **Exercise 5.1** For the graph in Fig. 5.2, what is the neighbourhood for the **Germany** node? What is the degree of this **node**? which **node** has the largest **degree**?, what **node** has the smallest **degree**?

✍ **Exercise 5.2** For the graph in Fig. 5.3, what is the **out-neighbourhood** of the **node** for **Cloudy**? What is the **out-degree** for this node? Which node has the largest **in-degree**? Is there a path from **Wet Grass** to **Cloudy**?

Note

A **graph** is an **ADT**. As we know, an **ADT** describes a way of organizing information, as well as a list of operations that must be supported by the **ADT**. In the case of **graphs**, the operations that must be supported include:

- Adding a **node**
- Removing a **node**
- Adding an **edge**
- Removing an **edge**
- Edge query (finding whether or not there is an **edge** joining two specific **nodes**) - this is sometimes called **adjacent**

In addition to the above, specific **graph** types may provide additional operations, such as finding the **neighbourhood** or the **degree** of a **node**.

5.1.4 Example applications of graphs

1) Network modeling: 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, determining the relative importance of individual nodes, and many other applications. A visual example of one such network is shown in Fig. 5.5.

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 **connected** 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

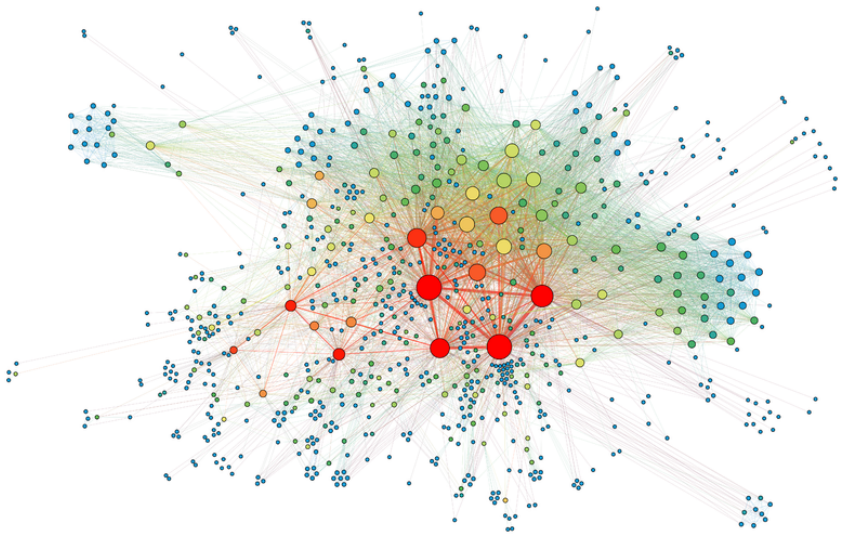


Figure 5.5: Social network analysis. Each circle is a **node** representing a person, the colour and size of the **node** indicate how **important** each node is in the **graph** - this is related to how well connected the node is, and whether its neighbours are themselves important.

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. An example of a document clustering graph is shown in Fig. 5.6.

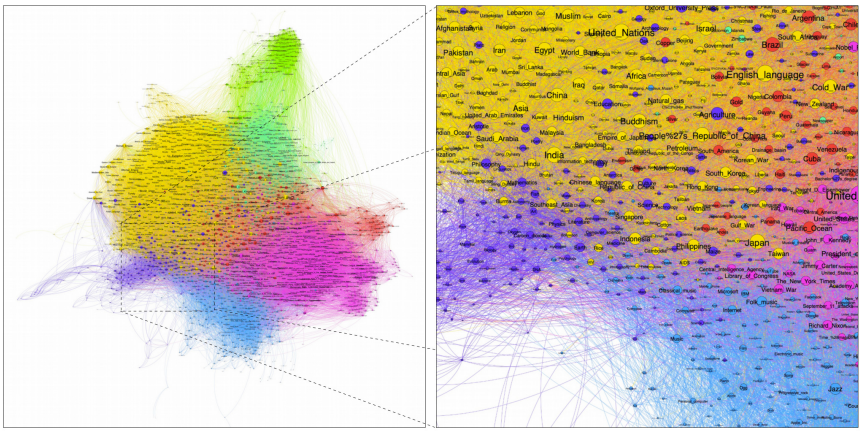


Figure 5.6: Top 2500 Wikipedia pages, grouped by similarity of content. Note that colour corresponds loosely to topic. *Image by Matt Biddulph, Flickr, CC-SA2.0.*

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. Fig. 5.7 shows a visualization of turbulence patterns for airplane modelling.

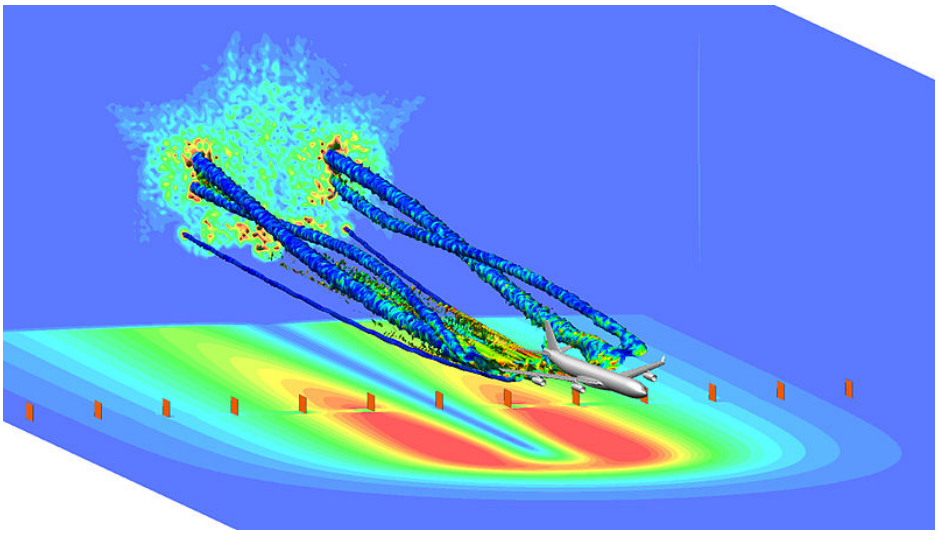


Figure 5.7: Simulation of the turbulence generated by an A340. A mesh of nodes (not visible in the image) distributed over the volume of this simulation forms the basis of the computation. The same mesh is used for visualization, by assigning a colour to nodes of interest. *Image: Deutsches Zentrum für 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. One such application is illustrated in Fig. 5.8 in the context of path finding for an aerial robot.

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. In this book, 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.

5.2 Representing graphs

There are several different ways of representing **graphs** (both un-directed and directed). They each have their own advantages and disadvantages, and the choice of method for any particular application will depend on how the graph will be used and must include a careful analysis of the **complexity** of performing the tasks the graph is meant to support. In what follows, we will consider two of the most often used methods for representing and managing **graphs**. Each method must solve two problems:

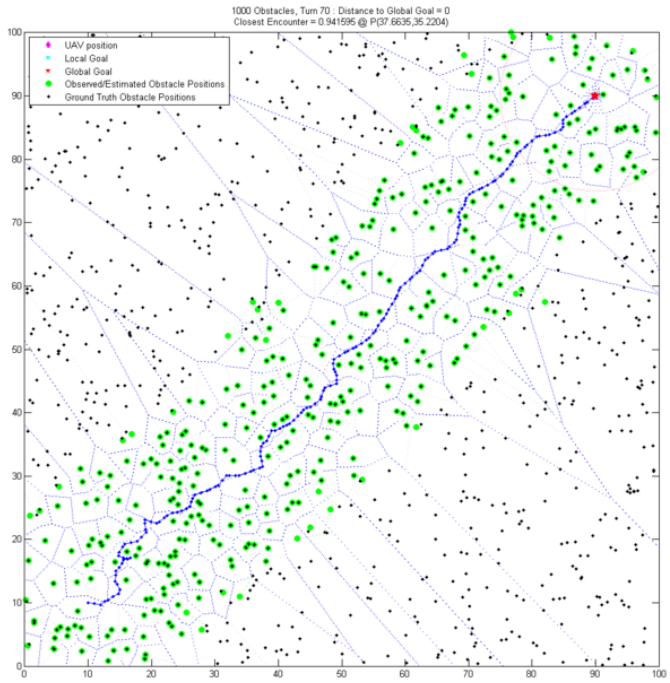


Figure 5.8: A path-planning simulation for an autonomous flying vehicle carrying a search and rescue mission in a forest. *Image: Elucidation, Wikimedia Commons, CC-SA3.0.*

First, storing and managing the set V of **nodes** that correspond to the data items in our problem. These can be stored using any of the data structures we have studied up to this point (e.g. arrays, linked lists, trees, etc.). The choice must be considered carefully, and has to be based on what **operations** will be performed on these data items, and the **complexity** of these operations.

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 directed graphs, the direction of each edge. The two methods we will discuss below deal with this particular problem in different ways - which will affect the **complexity** of performing all supported **graph operations**, as well as the **amount of space** required to store the edge information.

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 in Fig. 5.9. 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 the **edge query** require 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 $N \times N$, For **un-directed graphs**, entries $A[i][j]$ and $A[j][i]$ are both **1** if there is an **edge** joining **node i** and **node j**; it is 0 otherwise. For **directed graphs**, entry $A[i][j]$ is set to **1** if there is an edge **from i to j**, and is **0** otherwise. This is illustrated in Fig. 5.10. Note that for **un-directed graphs**, the **adjacency matrix** is **symmetric**.

Adjacency matrices have the same advantages and disadvantages of arrays: Edge queries are fast (no traversal required). Adding or deleting **edges**, and finding out whether two **nodes** are connected requires a single access to

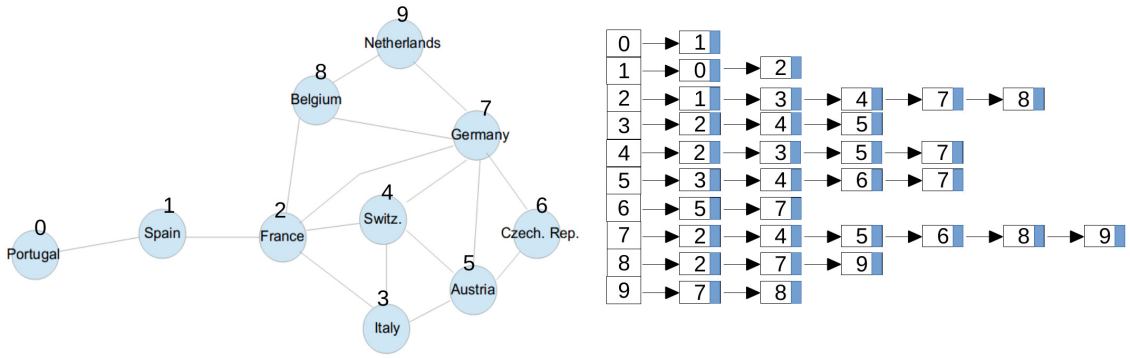


Figure 5.9: An adjacency list for the European countries **graph**. There is one entry per **node**, and each entry points to a linked list containing the indexes of the neighbours for that node. Therefore each entry in this linked list represents an **edge** in the graph.

the matrix. Conversely, they are not space efficient. Even in the small example in Fig. 5.10, 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 within the available memory.

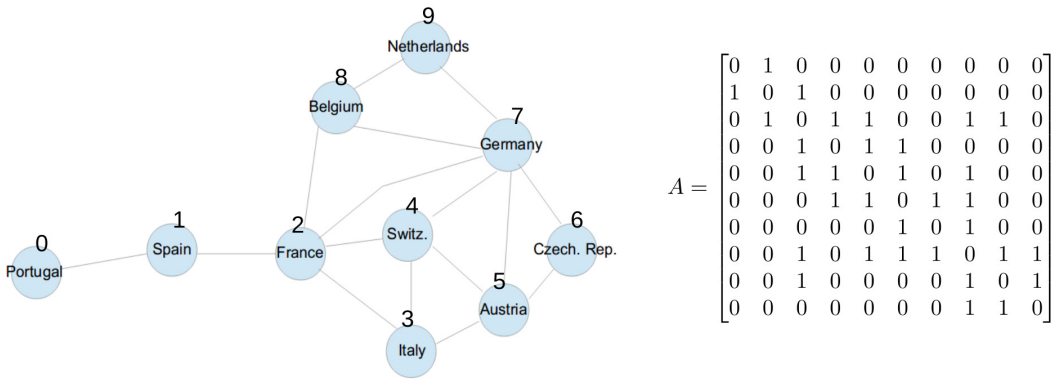


Figure 5.10: An adjacency matrix for the European countries **graph**. An edge is indicated by an entry in the matrix whose value is **1**, for example, $A[0][1]$ and $A[1][0]$ are set to **1** to account for the edge joining Portugal and Spain.

Note

For general **graph** applications, each **edge** may be associated with a **value** that represents **relevant information** about the relationship between the two bf nodes. For example in a **graph** that describes the **similarity** of songs, the **edge value** may represent **how similar** two songs are, with higher values indicating greater similarity and low values indicating the opposite.

If the **graph** has **weighted edges**, the **edge values** have to be stored. If using an **adjacency list**, each entry in the list will store **the index** of the connected **node** as well as the **edge value**. If using an **adjacency matrix** we simply store the **edge value** in the corresponding entry of the matrix. In this latter case, the **edge value** can not be zero (which indicates no connection).

✍ **Exercise 5.3** For the small example graph shown in Fig. 5.11:

- Show the **adjacency list**, use indexes $0-6$ corresponding to the nodes labeled $A-F$
- Show the equivalent **adjacency matrix** for this graph

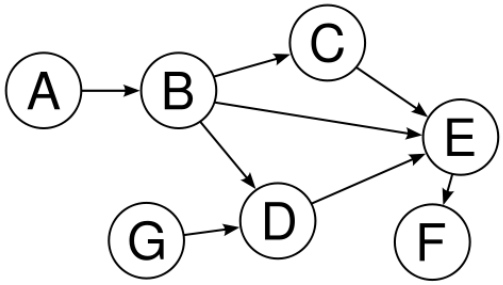


Figure 5.11: A small directed graph. *Image: David W., Wikimedia Commons, Public Domain.*

5.3 Complexity of Graph Operations

Let us consider the complexity of each of the operations the **graph ADT** specifies must be supported in the context of the two strategies discussed above for storing the **edge** information.

1) Adding an edge to the graph. Adding an edge to connect **node i** to **node j** has a **worst case complexity** of

- $O(1)$ for an **adjacency list** - we need to go to the entry for **node i**, and insert index **j** in the linked list found there. We can insert this index **at the head of the list** so the cost is $O(1)$ - if the **graph** is **un-directed** we also have to add index **i** to the linked list for **node j**.
- $O(1)$ for an **adjacency matrix** - we need to set entry $A[i][j]$ to **1**, with a cost of $O(1)$. For an **un-directed graph** we also have to set entry $A[j][i]$ to **1**.

2) Removing an edge from the graph. Removing an edge connecting **node i** with **node j** has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - we need to go to the entry for **node i**, and there we will need to **traverse the linked list** until we find the entry for **node j** which we will remove. The linked list for **node i** will have a length equal to **degree(i)** (the number of neighbours of **node i**), which in the worst case is $N-1$ (the node is connected to every other node in the **graph**). For an **un-directed graph** we have to repeat the process with the linked list for **node j** to remove the entry for **i**.
- $O(1)$ for an **adjacency matrix** - we have to set entry $A[i][j]$ to **zero** (and also entry $A[j][i]$ to **zero** if the **graph is un-directed**). These operations have a cost of $O(1)$ for the 2D array that stores the matrix.

3) Adding a node to the graph. Adding a new node has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - the list is an **array with size N**, so to add one node we have to **create a new array of size N+1** then we have to copy the N entries in the original array to the new one.
- $O(N^2)$ for an **adjacency matrix** - the matrix is an $N \times N$ array, so to add a node we have to **create a new array of size** $N+1 \times N+1$ and then copy the $N \times N$ entries from the original matrix to the new one. This is a computationally expensive operation for large graphs, and we also have to consider that we will need at least $2 \times N^2$ memory space to carry out the copying process, which may not be feasible. For this

reason **adjacency matrices** are appropriate only for cases in which the **graph** isn't changing frequently, or for smaller graphs.

4) Removing a node from the graph. This has **worst case complexity** of

- $O(M)$ for an **adjacency list** - to remove a **node**, we have to remove all **edges to and from this node** in the **adjacency list**. This requires traversing all linked lists of edges and removing any that correspond to the **node we are removing**. There are M entries in these lists for a **directed graph**, or $2M$ entries for **un-directed graphs**. In the worst case, for a **fully connected graph**, $M = N^2$ so it is also correct to state that the **worst-case complexity** of removing a node when using an **adjacency list** is $O(N^2)$.
- $O(N)$ for an **adjacency matrix** - removing all edges **to and from a node** requires us to set to **zero** all the entries in the **row** and **column** corresponding to that node. This involves visiting $2N$ entries in the **adjacency matrix**.

Note

In the above, the actual **adjacency matrix** or **adjacency list** are not resized - this is the most common way to implement **node** removal because it allows all other existing nodes to **keep their current index**. Remember that the actual **data items** for each of the nodes in the **graph** are stored in a separate data structure - and the **node indexes** connect our **adjacency list** or **adjacency matrix** to entries in this data structure so we have access to the relevant data when we need it.

5) Edge query. Figuring out whether there is an edge connecting **node i** to **node j** has a **worst case complexity** of

- $O(N)$ for an **adjacency list** - we have to **traverse the linked list** for **node i** and see if we can find an entry for **j** in this list. The list has a length of $degree(i)$ (the number of neighbours of **i**). I the worst case, for a fully-connected graph, $degree(i) = N - 1$.
- $O(1)$ for an **adjacency matrix** - we can directly check the value of entry $A[i][j]$ in the **adjacency matrix** and see if it is non-zero.

✍ **Exercise 5.4** Figure out and explain the **worst case complexity** of finding the **degree of node i** ($degree(i)$) using an **adjacency list** and using an **adjacency matrix**. This is for an **un-directed graph**.

✍ **Exercise 5.5** Figure out and explain the **worst case complexity** of finding the **in-degree of node i** ($in - degree(i)$) using an **adjacency list** and using an **adjacency matrix**. This is for an **directed graph**.

Now that we know how to represent and store a **graph**, and have looked at the **complexity** of the fundamental operations we can perform on **graphs**, we are ready to start thinking about the kinds of problems we can solve using them. However, before we can really explore interesting applications of **graphs**, we first need to study a general **problem solving technique** that is very often used together with **graphs**, as an essential component of algorithms that use the **graph** to process information: **Recursion**.

5.4 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 for some time. Recall we made the point in the text above that both **linked-lists** and **BSTs** are **graphs**. And we just said that **recursion** is strongly tied to algorithms that work on **graphs**.

As we noted at the time, 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. 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**, which follows from the structure of the tree (remember we noted that every sub-tree of a **BST** is also a **BST**).

What we will do now is develop a general picture of what **recursion** is

- A way of thinking about problems that have particular structure
- A way to implement programs that solve such problems
- A tool for simplifying complex tasks that are difficult to handle otherwise

5.4.1 Examples of recursive problems and data structures

1) Graph search: **Graphs** are a **recursive** form of data representation. Every **sub-graph** of a **graph** is also a **graph** as illustrated in Fig. 5.12.

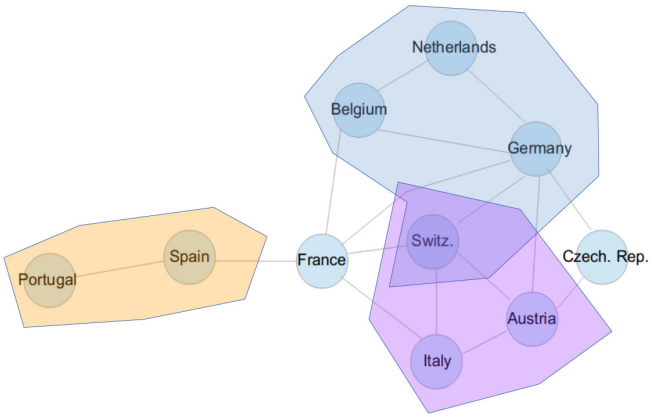


Figure 5.12: A **graph** and several possible **sub-graphs** (in colour), each **sub-graph** is itself a **graph** with its sets V of nodes and E of edges.

We had already seen this property with **BSTs**, but now we know it is 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
     Solve the smallest problem and use that solution to solve the slightly larger one
   And then the even larger one
 And then the even larger one
Until we have the solution to the original complex problem
```

This general process for looking for a specific **node** in a **graph** is an example of **recursion**. Variations of the process described above are used for **path finding**, **robot activity 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 in a simulation, can be represented by a **graph** with **one node per location**, and **edges** linking together neighbouring locations that are connected (i.e. one can move from one such location to the next, there are no barriers in between).

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 8×8 map, each location has been assigned a **node** in a **graph**, and the **nodes** are connected if the corresponding map locations are neighbours and there are no walls in between them. This is illustrated in Fig. 5.13.

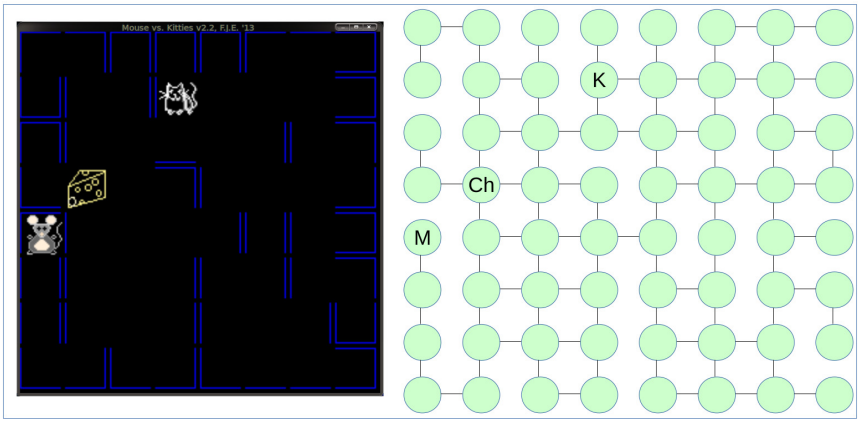


Figure 5.13: An example of a **graph** representing a small 8×8 maze. Each **node** corresponds to a location in the map, **edges** link together neighbouring locations that are connected (no walls between them), and represent the fact that someone walking around this maze could move from one location 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 5.6** Develop **pseudo code** to solve this problem using only **iteration** (loops) but **no recursion**. You must follow the problem solving process we developed in Chapter 2, fully understand the problem, and develop a solution

that is clear and detailed enough to serve as a basis for implementing a program to solve this task. Specifically, you should carefully consider how the **graph** will be processed (the order in which **nodes** will be considered, how to get information about neighbours, etc.) as well as **ow to keep track** of any information your process needs to form a complete path from the mouse to the cheese.

If you spend any time at all solving the previous exercise, you will 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 ADT** by nature, you **should expect that non-recursive algorithms working on graphs will be cumbersome** and difficult to implement, test, and maintain. Conversely, the **recursive** solution is **simple**, **elegant**, and **easy to implement**.

The process is illustrated in Fig. 5.14 and we will take a detailed look at how it works later on in this Chapter.

2) Divide and conquer methods: Divide-and-conquer methods are behind some of the most powerful and useful algorithms 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 $O(NLog(N))$, 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.

For example, mergesort is a sorting algorithm guaranteed to sort an input collection with complexity $O(NLog(N))$. The method works as follows (pseudo code):

```
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
```

We will look at this process in detail later on in this Chapter.

3) Computer graphics: Recursive structures are common in computer graphics, for example:

- Objects are often composed of parts (e.g. the various limbs of a person or animal, and different parts of plants or buildings). These are often modeled using tree-like structures - that brings us back to **graphs** which have recursive structure
- The rendering (drawing) process for certain plants like ferns and trees is naturally recursive because it involves drawing the same shape but at different sizes. This is shown in Fig. 5.15.

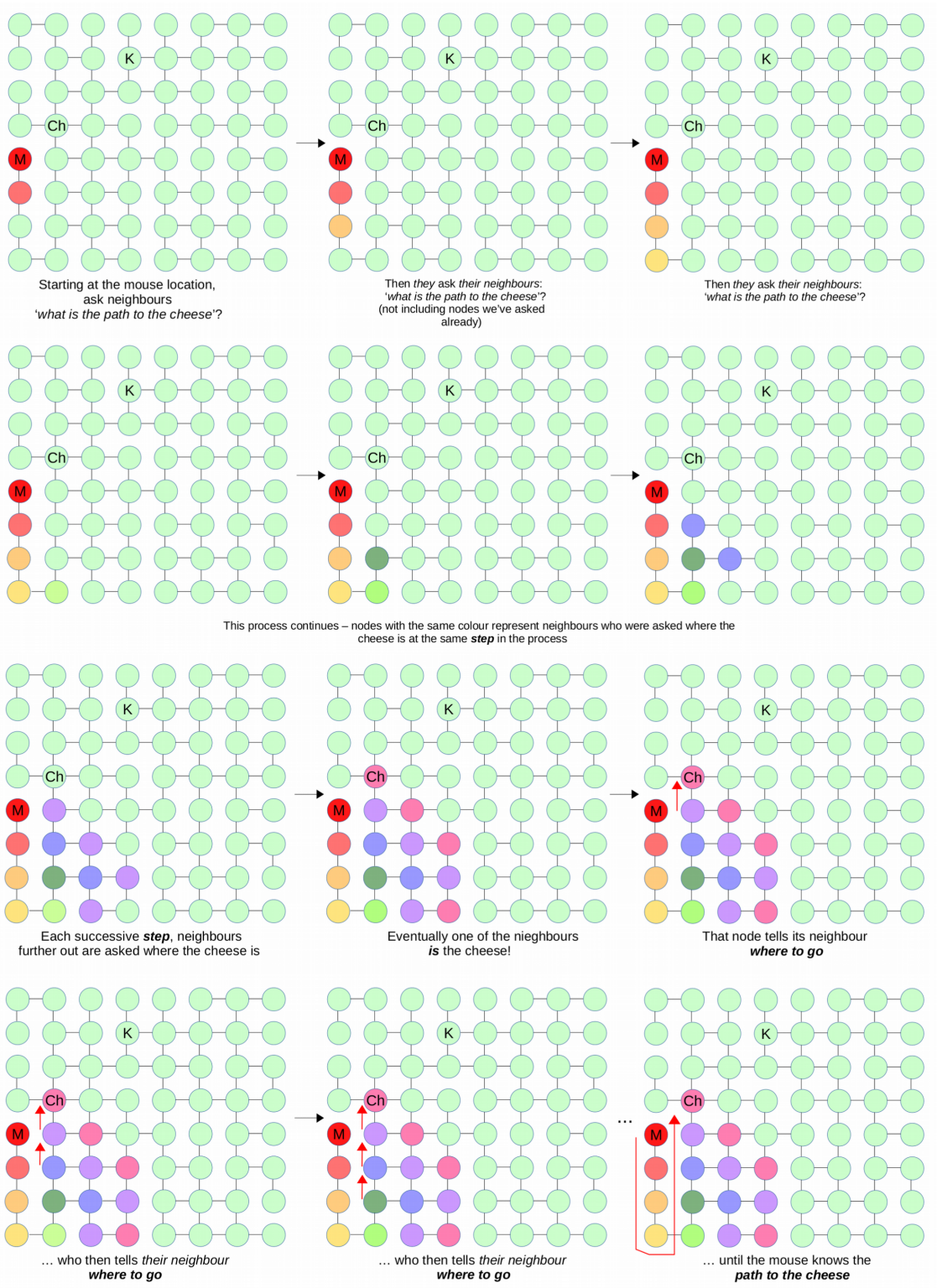


Figure 5.14: Illustration of the **recursive** path-finding process for the maze example. At each step, nodes farther and farther away from the mouse are asked to find a path to the cheese. Eventually, one of the nodes is a neighbour of the cheese and knows where to go. The information then propagates backward neighbour to neighbour until the full path from the mouse to the cheese is obtained.

- 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.

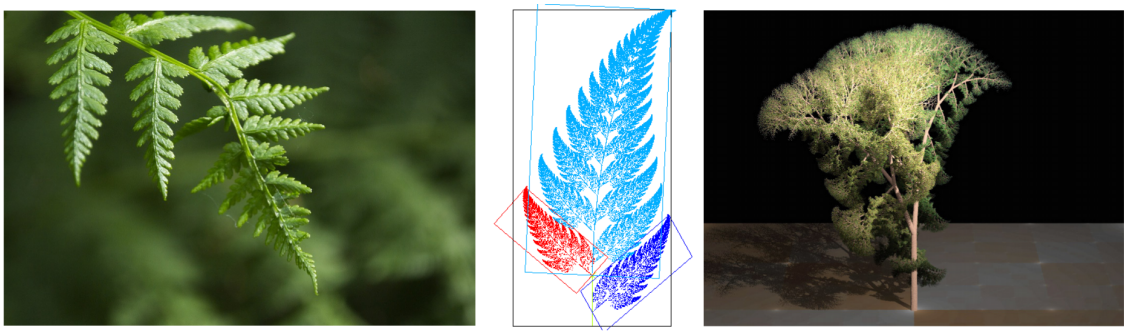


Figure 5.15: A real fern (left), a computer model of a fern (center) showing how the fern's parts are just smaller versions of the whole shape, and a computer rendering (right) created using a tree-based representation and a recursive rendering algorithm to create a life-like virtual plant. *Images: (left) Pixbay, used with permission, (center) A. M. Campos, Wikimedia Commons, public domain, (right) Solkoll, Wikimedia Commons, public domain.*

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.

4) 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, but change the syntax used as well as the features they offer to make your work as a developer easier.

The programs that **parse** and **compile** programs use **graph** representations and **graph methods** for representing the structure of instructions in the program, checking them for syntactical correctness, determining their meaning, and generating the relevant code. These tasks are often solved with **recursion**.

A different class of programming languages follow what is called a functional programming model (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. **Recursion** is often at the centre of such languages and is a fundamental part of how we have to think about problems if we want to approach them using functional programming.

5) File-system organization: As a final example of **recursive problems** and data structures, consider the organization of the file system in your computer. The information you have stored there is organized into folders, each of which will contain multiple items which themselves can be folders. The **directory structure** in the file

system is in fact a tree (**graphs** again!), and is recursive. How would we go about finding a file, if we do not know in which folder it is stored? a general **recursive** solution would look like so:

```
// Find the name of the folder in which the requested file is stored

findFile(directory, file name)
  if a file matching 'file_name' is in 'directory'
    return 'directory'
  else
    for every 'sub_directory' in 'directory'
      directory_name = findFile(sub_directory, file_name)
      if 'directory_name' is not [empty]
         return 'directory_name'

      return [empty]
```

The above process travels through the directory structure in the computer, starting at an initially specified directory, 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 until either the file is found, or the entire directory structure in the 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) as shown in Fig. 5.16.

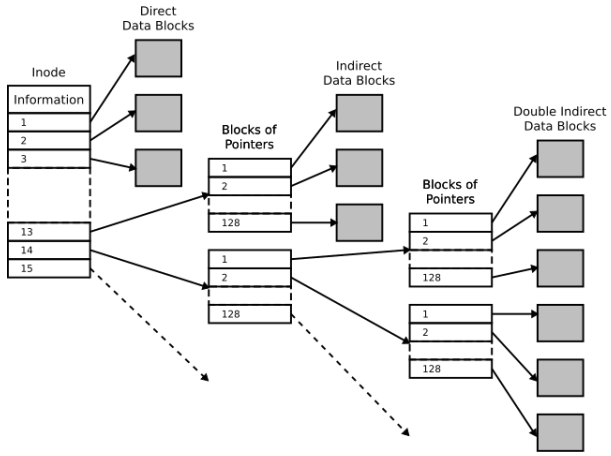


Figure 5.16: Structure of the data component of a Linux EXT2 File System **inode**. *Image: timtjtim, Wikimedia Commons, CC-SA4.0.*

5.5 General principles of recursion

By now you should have general idea of the kinds of problems that are suitable for solving with **recursion**, as a summary, we should consider using **recursion** to solve a problem when:

- The problem itself is complex - if there is a simple, direct solution using loops, you should go for that instead.
- The problem can be broken down into smaller or simpler versions of itself - e.g. a large **graph** can be broken up into smaller and smaller **subgraphs**.

- The nature of the problem is the same regardless of the size of the input - e.g. the problem of finding a path between two **nodes** is the same regardless of the size of the **graph**.
- At some point, the solution to one of the smaller problems becomes easy to compute - this allows us to stop breaking the problem down.
- We can use solutions to smaller problems to build the solution for larger problems easily.

This is fairly informal, so let us take a look at the components and process involved in working out a recursive solution to a problem.

5.5.1 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) The Base Case: This corresponds to the simplest (trivial) form of the problem we are solving, we should be able to easily and directly solve the problem for the base case without having to break the problem down any further. Every recursive solution **must have** at least one **base case** otherwise the recursive process keeps endlessly trying to simplify the problem and never returns a solution.

Example 5.1 Some examples of **base cases** for common applications of recursion include:

- For problems involving strings, the **base case** often is an empty string, or a string with one character - whatever the task, we can easily carry it out on a single character, and there is often nothing to do for the empty string.
- For numeric arrays, the **base case** often is an array with 1 entry, or an empty array - for instance, sorting an array with a single entry is trivial.
- 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 - for instance, for mapping applications, it could be that the node being processed represents the location we are looking for.
- For information search (on lists, trees, or other data structures) the **base case** can be an empty data structure, or having found the node with the information we need.

Note

As you can see there can be several different **base cases** for a given problem. It is important to **consider all the possible base cases** that apply. In the examples above we have listed a few of the common **base cases** for a few common problems. But notice that we only say these apply **often**. For a particular problem, they may not apply, or there may be additional **base cases** not listed above. The importance of making sure all the relevant **base cases** have been accounted for is that, when one or more of them are missing, there would be inputs for which our recursive solution would not work. It would reach one of the missing **base cases** and just keep going endlessly (which in practice often results in the program crashing).

2) The Recursive Case: This corresponds to the general form of our problem, which we have to split, simplify, or otherwise make smaller in order to get one of the **base cases**. The interesting part of solving a problem with **recursion** involves thinking about how to break down our specific problem. After we have split the problem in an

appropriate way into two or more sub problems, we then **recursively solve** the smaller sub problems. The **recursive case** implementation is also responsible for **building a solution** for the original problem from the solutions to the separate sub problems we split it into.

Example 5.2 Here are a few examples of how different types of problems can be broken down

- For strings, the **recursive case** often consists of breaking 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, if we keep splitting each chunk we will eventually reach the **base case**.
- For numeric arrays, the **recursive case** 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 subgraphs - for example, in the path-finding problem discussed above, the process considers at each step only a small set of **neighbours** that have not yet been checked for a path to the destination. Another possibility is that the **recursive case** may split the graph into disjoint subsets 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 subtrees, and processes these recursively until reaching the base case of an empty subtree.
- For search, the **recursive case** often involves searching over a subset of the data structure. For example, for **BSTs**, 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).

Note

It is essential for the **recursive case** to bring the problem closer to a **base case**. If the problem is not getting easier to solve despite repeated application of the **recursive case**, or if it takes too many steps to reach a **base case**, we should carefully re-think our process. As we shall see in short order, the choice of how to break down a problem for the **recursive case** can have a significant impact on the **computational complexity** of our **recursive** solution.

5.5.2 How to design a recursive solution

The first step of the process is exactly as discussed at the end of Chapter 2 - thoroughly understand what the problem is, write down and illustrate an example - e.g. if you are working on arrays, draw a representative sample array with data, and consider if the problem has the requisite properties that make it a good candidate for a **recursive** solution.

Then develop the algorithm **in pseudo code** as per the discussion in Chapter 2, paying special attention to:

- What are the the **base cases** for this problem? It is important to work out **on paper** a few examples of how small versions of the problem may look, and ensure that no **base cases** are missing.

- Once you we have the **base cases**, determine how you could split an instance of the problem that is **not a base case** into **smaller or simpler sub problems** that will get closer to the base case - this gives you the **recursive case**. At this point it is also important to work out how to build the solution for the original problem from the solutions to smaller sub problems. Once more **work out a few examples on paper** and make sure the process works, this will often allow you to find any missing **base cases**.
- Consider different ways of splitting the problem for the **recursive case**. Try to figure out if different choices make your solution do more or less work in terms of the number of times the **recursive case** needs to be applied, and in terms of the **complexity** of building bigger solutions from those for smaller sub problems.
- Your pseudo code algorithm should clearly involve **recursively solving sub problems**.

Note

Recursive solutions that have not been carefully constructed can be difficult to **test**, **trace**, **debug**, and **maintain**. So special care has to be taken to ensure the proposed algorithm is **clean - with reasonable and easy to understand base and recursive cases**, **carefully thought out**, and **well documented** - this means carefully noting design choices and your rationale for how the **recursive case** was selected.

Let us see how the process described above can be applied to solve a very common problem: **sorting** a large collection of information.

5.5.3 Sorting data recursively

Let us consider the problem of **sorting data stored in an array**. In the discussion below the array contains **integers** but this is not a limitation, any data for which we can implement a meaningful **comparison function** can be sorted in the same way. Our goal will be to develop a **recursive** solution by following the process described above, and to study the effect of different ways to implement the **recursive case** on the **computational complexity** of the resulting sorting algorithm.

The Problem: Given an input array of **un-sorted** data, return an array with the same number of entries **sorted** in ascending order. We will work with the following sample array:

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

this is a very small array, but it is enough for us to understand the problem and develop our **recursive** solution.

Step 1) Find the base cases: We can figure out what these should be by considering the easiest instance of our problem. For **sorting**, it doesn't get any easier than having an **empty array** - there is nothing to do, since it contains no data. It is a valid **base case** and we will see shortly that it comes up during the process of sorting non-empty arrays. There is another **base case** corresponding to an array with a single entry. This is a base case because a single-entry array is **already sorted**.

Any array that has more than one entry may need to be sorted, it can also be split into smaller arrays that are closer to one of the two **base cases** above. Therefore **arrays with more than 1 entry are not base cases**.

2) 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 noted 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 study the complexity of the sorting process for each of the splitting methods, and realize that seemingly small differences in how the **recursive case** works can have a large effect on **complexity**.

```
Algorithm: sort_case1()

// Recursively sort an array of integers

Input: An 'array' of length N containing integers
Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' and 'sub2' such that:

          // Recursive case #1: split the array into first entry and everything else

          sub1=array[1]             // First entry in the array
          sub2=array[2:end]         // All remaining entries in the array

      Recursively sort 'sub1' and 'sub2' using recursive case #1

          subsort1=sort_case1(sub1)
          subsort2=sort_case1(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

          sort=merge_in_order(subsort1, subsort2)
          return 'sort'
```

Question: Does the **recursive** case work? does it eventually reduce an array of any size to sub-arrays that are one of our two **base cases**?. The splitting process will be applied to any array of size 2 or greater. It will split the array into two **smaller** sub-arrays. These will undergo further splitting as needed. At each step the resulting sub-arrays are smaller, so eventually we must arrive at one of our **base cases**.

Let's see the process in action, on the small input array shown above:

```
[ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]                 // Original array

[9] : [6 | 3 | 7 |0 | 2 | 8 | 1 | 4 | 5 ]                 // First split

// [9] is in base case, but we have to wait for the second
// sub-array to be sorted in order to re-build the complete
// sorted array

[9] : [6] : [3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]               // Second split
```

```
// [6] is in base case, but we need to wait for the second
// sub-array to be sorted... [9] is still waiting!

[9] : [6] : [3] : [7 | 0 | 2 | 8 | 1 | 4 | 5 ]            // Third split

// [3] is in base case, need to wait for the second sub-array
// to be sorted. [9] and [6] are still waiting for a result

.
.      // splitting continues...
.

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4 | 5]    // Eight split

// 9, 6, 3, 7, 0, 2, 8, and 1 are waiting, need to sort
// [4 | 5] which is not in base case

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4] : [5] // Final split

// [4] and [5] are base case, so the recursive calls simply
// return the same two sub-arrays and they can be merged
// (they are merged in sorted order)

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1] : [4 | 5]    // First merge

// [1] was waiting for [4 | 5] to be sorted, it is now
// sorted so it gets merged with [1]

[9] : [6] : [3] : [7] : [0] : [2] : [8] : [1 | 4 | 5]      // Second merge

// [8] was waiting for [1 | 4 | 5] to be sorted, it is
// now sorted so it gets merged with [8]

[9] : [6] : [3] : [7] : [0] : [2] : [1 | 4 | 5 | 8]        // Third merge

.
.      // merging continues...
.

[9] : [6] : [0 | 1 | 2 | 3 | 4 | 5 | 7 | 8]                // Seventh merge

// [6] was waiting, it can now be merged into a larger
// sorted array

[9] : [0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]                 // Eight merge

// Finally, [9] was waiting, and it can be marged with the
// remaining (now sorted!) entries to produce the final
// sorted array

[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]                   // Final merge
```

As you can see, the process itself is fairly straightforward. The array is reduced to individual entries one element at a time, until all the sub-arrays are in **base case**, then they are merged in order one by one to build the final sorted array.

Note

Merging two sorted sub-arrays with **N** entries in total into a single sorted array requires $O(N)$ time. The process is as follows:

```
// Procedure merge_in_sorted_order()
// input: two sorted sub-arrays 'sub1[]' and 'sub2[]'
// output: a single 'sorted[]' array containing the entries from both sub-arrays

out_index=0;      // Indexes into the output array and each
sub1_index=0;     // input sub-array. Initially 0 (first entry)
sub2_index=0;

while out_index is less than N      // This loops exactly N times

    choose whichever entry is smaller between

        sub1[sub1_index] and sub2[sub2_index]

    copy the chosen entry to sorted[out_index]

    if the entry from sub1 was selected, increment sub1_index by 1
     otherwise, increment sub2_index by 1

    increment out_index by one
```

Complexity of sorting using recursive case #1: Let's see how much work needs to be done by the sorting process when we split arrays according to **recursive case #1**.

- The **recursive case** has to perform **N-1** splits and **N-1** merges - this is because each split except for the last one leaves a single entry of the array in the **base case** (the last split takes care of two entries). So there is a factor of **N** here.
- For an array of size **N**, the split has a cost of $O(N)$ because **N** elements have to be placed in sub-arrays (copying the **N** entries into sub arrays requires **N** copy operations).
- Combining two **sorted** sub-arrays with **N** entries (combined) into a single **sorted** array of size **N** has a cost of $O(N)$.
- Therefore, each split together with the corresponding merge has a cost of $O(N)$.
- Since there are **N-1** split/merge steps, each with a cost of $O(N)$, the total cost for the sorting process is $O(N^2)$.

This means that the recursive sorting algorithm based on **recursive case #1** is just as slow as **bubble-sort**. This is not a good result. Looking at the example above with the 10-entry array, we note that it takes **a lot of splitting and merging** to get the job done. This is because **recursive case #1** reduces the size of the problem by **1** at each step. What can we do to reduce the number of steps required to reduce the sub-arrays to a **base case**?

```
Algorithm: sort_case2()
```

```
// Recursively sort an array of integers

Input: An 'array' of length N containing integers
Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' and 'sub2' such that:

          // Recursive case #2: split the array in two halves

          m=floor(N/2)               // Half the size of the input array, rounded down.
          sub1=array[1:m]            // First half of the array
          sub2=array[m+1:end]        // Second half of the array

      Recursively sort 'sub1' and 'sub2' using recursive case #1

          subsort1=sort_case2(sub1)
          subsort2=sort_case2(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

          sort=merge_in_order(subsort1, subsort2)
          return 'sort'
```

Let's see how the above process works on the same input array we used with **recursive case #1**:

```
  [ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]              // Original array

                    (A)
  [9 | 6 | 3 | 7 | 0] : [2 | 8 | 1 | 4 | 5]              // First split

  // Both arrays are not base case, each has to be sorted using
  // recursive case #2, this means 9,6,3,7,0 will be split in
  // half, and 2,8,1,4,5 will be split in half (splits are
  // labeled B for clarity)

        (B1)                    (B2)
  [9 | 6] : [3 | 7 | 0] : [2 | 8] : [1 | 4 | 5]           // Second split

  // All the sub-arrays are still not base case, so each will be
  // sorted using recursive case #2 and split in half again
  // (splits are labeled C for clarity)

     (C1)        (C2)            (C3)        (C4)
  [9] : [6] : [3] : [7 | 0] : [2] : [8] : [1] : [4 | 5]    // Third split

  // The splits C2 and C4 still have one sub-array (each) that
  // is not a base case, so one more round of recursive case #2
  // is applied (with splits labeled D for clarity)

                       (D1)                          (D2)
  [6] : [9] : [3] : [7] : [0] : [2] : [8] : [1] : [4] : [5] // Final split
```

```
// Reached base case for all sub-arrays, now we just merge
// in the reverse order - first the D splits merge

[6]: [9] : [3] : [0 | 7] : [2] : [8] : [1] : [4 | 5]      // First merge

// Now the C splits merge

[6 | 9] : [0 | 3 | 7] : [2 | 8] : [1 | 4 | 5]             // Second merge

// Now the B splits merge

[0 | 3 | 6 | 7 | 9] : [1 | 2 | 4 | 5 | 8]                // Third split

// And finally the A split merges to create the final ourput array

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

You can probably already see that **recursive case #2** takes fewer steps to reach the **base case** for all the sub-arrays. Since at each step, the size of the arrays is halved, the number of steps required to reach the **base case** is $Log_2(N)$ - the progressive halving of the size of the input is the same process that allows **binary search** to achieve $O(Log(N))$ search complexity and that allows **BSTs** to (on average) provide $O(Log(N))$ complexity for search, insert, and delete.

Complexity of sorting using recursive case #2: Let's see how much work needs to be done by the sorting process when we split arrays according to **recursive case #2**.

- The **recursive case** has to perform $Log_2(N)$ splits and $Log_2(N)$ merges - this is because each split reduces the size of the input by a factor of 2, and each merge produces a sorted array twice the size of the input sub-arrays.
- For an array of size **N**, the split has a cost of $O(N)$ because **N** elements have to be placed in sub-arrays (copying the **N** entries into sub arrays requires **N** copy operations).
- Merging two **sorted** sub-arrays with **N** entries (combined) into a single **sorted** array of size **N** has a cost of $O(N)$.
- Therefore, each split together with the corresponding merge has a cost of $O(N)$.
- Since there are $O(Log_2(N))$ split/merge steps, each with a cost of $O(N)$, the total cost for the sorting process is $O(N \cdot Log(N))$.

This is a great result - the sorting algorithm that uses **recursive case #2** has a guaranteed **worst case complexity** of $O(N \cdot Log(N))$. You will recognize that this is in fact **merge-sort**, a solid, reliable, and easy to implement method that shows the power of **recursion**. It should give you a moment of pause to consider that the two sorting algorithms we discussed thus far are **identical** save for the **choice of recursive case**. Splitting the array in two rather than one-versus-the-rest gives us an algorithm whose complexity matches the best known complexity for general sorting algorithms.

There is one more choice of **recursive case** that is worth thinking about:

```
Algorithm: sort_case3()

// Recursively sort an array of integers

Input: An 'array' of length N containing integers
Output: A 'sorted' array of integers of length N

If 'array' is a base case: {empty array, array of size 1}

      return 'array'    // Base case - it's already sorted!

Otherwise

      Split 'array' into 'sub1' 'pivot' and 'sub2' such that:

          // Recursive case #3: split with regard to a pivot entry

          choose m=random entry in 0:N-1 // Select a random entry
          pivot=input[m]                 // The pivot is the value
                                         // of this entry
          sub1 := All entries in input whose value < pivot
          sub2 := All entries in input whose value > pivot

      Recursively sort 'sub1' and 'sub2' using recursive case #3

          subsort1=sort_case3(sub1)
          subsort2=sort_case3(sub2)

      Merge the sorted sub-arrays in order to build the complete
      sorted array corresponding to the input

          sort=merge_in_order(subsort1, pivot, subsort2)
          return 'sort'
```

This process doesn't split the input into sub-arrays of a pre-defined size. Instead, it chooses a random entry in the input array (called the **pivot**) and then splits the input into two sub-arrays consisting of the elements that are **lesser than the pivot** and the entries with value **greater than the pivot**, plus the **pivot** itself (notice this is a three-way split). The splitting process **does not sort the entries in each sub array** - but it does **sort the input values with regard to the pivot**.

Let's see how **recursive case #3** works on the sample input array:

```
[ 9 | 6 | 3 | 7 | 0 | 2 | 8 | 1 | 4 | 5 ]                    // Original array

    m=2, pivot=3 (input[2]=3)
              (A)
[ 0 | 2 | 1] : [3] : [9 | 6 | 7 | 8 | 4 | 5]                  // 1st split

// Notice the sub-arrays are not the same size, and that
// within each sub-array the entries are not sorted, but
// each set is sorted w.r.t. the pivot. The pivot is
// now in the correct sorted spot!
// The two sub-arrays are not base-case, we have to
// apply recursive case #3 to each of them (the splits
// are labeled B for clarity)

    m=1, pivot=2          m=5, pivot=5                        // 2nd split
```

```
          (B1)                  (B2)
[0 | 1] : [2] : [] : [3] : [4] : [5] : [9 | 6 | 7 | 8]

// Each split will have a different pivot. We still have
// several arrays not in base case, so apply
// recursive case #3 again (splits labeled C for
// clarity - pivot is under the label)

     (C1)                                                (C2)
[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6 | 7] : [8] : [9] // 3rd split

// Need one more round of recursive case #3 to bring
// everything to base case (splits labeled D for
// clarity, pivot is under the label)
                                                    (D1)
[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6] : [7] : [] : [8] : [9] // Done!

// At this point we are done splitiing, merge sub-arrays
// in reverse order. Start with D splits

[] : [0] : [1] : [2] : [] : [3] : [4] : [5] : [6 | 7] : [8] : [9] // 1st merge

// Then the C splits

[0 | 1] : [2] : [] : [3] : [4] : [5] : [6 | 7 | 8 | 9]            // 2nd merge

// Then B splits

[0 | 1 | 2] : [3] : [4 | 5 | 6 | 7 | 8 | 9]                      // 3rd merge

// And finally the A split to yield the complete sorted result

[0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9]                         // Final merge
```

This sorting method is called **quicksort** and is one of the most commonly used, and most reliable general sorting algorithms. As the name implies, it is often **fast**, but it's not clear just from the example above why that would be the case.

However, you may be able to see a few features of **quicksort** in the example above: The splits are not symmetric - they can happen anywhere along the array. Sometimes they are close to the **one-versus-the-rest** case, sometimes they are closer to the **split-in-half** case, and sometimes we even have an empty sub-array on one side or the other from the **pivot**. It all depends on luck since we choose the **pivot** randomly for each split. You can also see that **recursive case #3** appears to do less splitting and merging than **recursive case #1**, but it's not clear just how much more or how much less work it does compared to **merge sort**.

So the question is: What can we say about the complexity of sorting with **quicksort**?

In the **worst-case**, we get very unlucky with **each and every split** and choose the **pivot** that corresponds either to the smallest, or the largest value in the array. The result is one empty sub-array, the 1-entry array with the **pivot** itself, and one sub-array that contains the rest of the entries in the original. This is identical to **recursive case #1**, and results in an algorithm that performs **N-1** splits and **N-1** merges, with a computational cost of $O(N^2)$. The **worst case complexity** of sorting an array using **quicksort** is $O(N^2)$.

That doesn't seem good - why is **quicksort** so popular, so often used, and said to be **fast** when we have just

seen that it's **worst case complexity** is just as bad as **bubble-sort**?

The usefulness of **quicksort** results from what happens **in the average case**. The question we need to ask is **what is the probability that we hit the worst-case split**? For an array of size **N**, choosing the smallest or largest value by random chance has a probability of $2/N$. For a large **N** this is very very small! - of course, we could worry about splits that are close to the worst case, even if not the very worst, but it turns out even if we pick the smallest 100, and the biggest 100 pivot values, the probability of randomly selecting one of them is $200/N$ which again, for a large value of **N** will be very small.

Not only that, but the **worst case complexity** of **quicksort** happens when we consistently get the worst split - at every step of the sorting process. The probability of that happening with **random pivot selection** is astronomically small for a large input array.

So in practice, we don't have to worry much about encountering the **worst case** for **quicksort**. But that doesn't tell us anything about what to expect **on average**. The mathematical analysis of **quicksort** is beyond the scope of this Chapter, however, we can gain an understanding of why **quicksort** works so well by studying the results of simulating **quicksort** millions of times (with different, randomly generated, unsorted input arrays) for different values of **N**. Fig. 5.17 shows the results of simulating the **quicksort** splitting process $1,000,000$ times for arrays of size $N = 10,000$ and **counting the number of splits required to reach base case for all sub-arrays**.

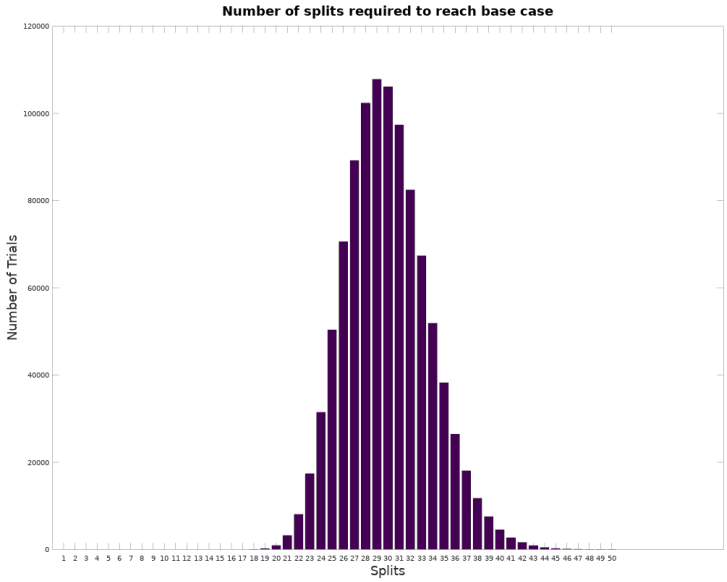


Figure 5.17: Number of levels required to reduce arrays of size $N = 10,000$ to **base case**. One million trials with different random arrays were simulated for this graph.

What we can see from Fig. 5.17 is that while the number of splits is **not** $Log_2(N)$ which for $N = 10,000$ is just over 13 splits, the maximum number of splits that were required over the entire set of one million different arrays is still pretty small (close to 50 which is tiny compared to the value of **N**). This indicates that **quicksort can reach the base case for all sub-arrays quickly**, without too many splits.

An even more interesting and encouraging picture emerges once we simulate the **quicksort** splitting process for different values of **N**, and plot the **average number of splits** required to bring all the sub-arrays to **base case**. This is shown in Fig. 5.18.

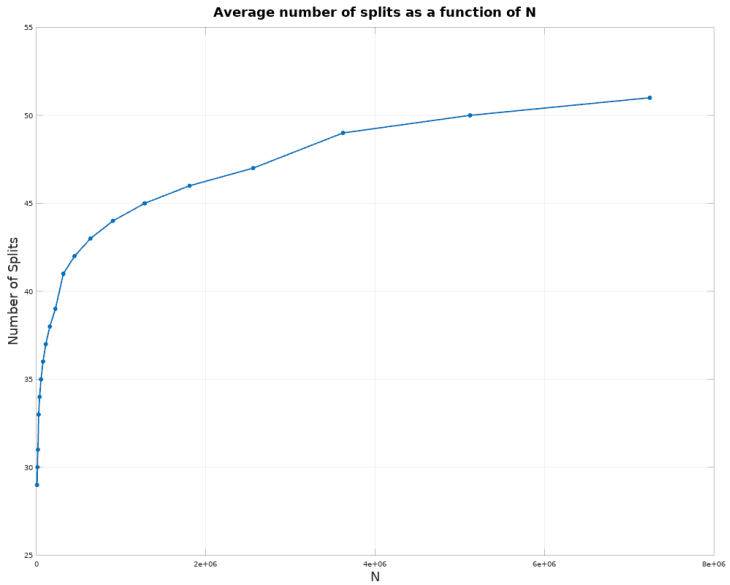


Figure 5.18: Average number of splits required to reduce arrays of size N to **base case** for N from $10,000$ to just under $8,000,000$. One million trials with different random arrays were simulated for each value of N in the graph.

The shape of the graph in Fig. 5.18 should be familiar - it is a **logarithmic** curve, and illustrates a well known result: The **average case complexity** of **quicksort** is $O(N \cdot Log(N))$. This is because **on average**, **quicksort** will carry out a number of splits and merges proportional to $Log(N)$, and each of these has a cost of $O(N)$. This is a very good result. In the **average case**, **quicksort** has the same complexity as **mergesort** and achieves the best known complexity for general sorting methods.

Note

You may be concerned that **quicksort** does more work during the splitting step than **mergesort** - it needs to compare each entry in the input array against the pivot and decide which sub-array it should go into. Conversely, **mergesort** just splits the arrays in half without additional work.
However, the extra work **quicksort** does when splitting evens out during the merging step - you can see in the example above that merging sub-arrays sorted with **quicksort** is easy because all the entries are in the correct location, **no additional comparisons are needed**. Conversely, **mergesort** has to compare entries from the sorted sub-arrays during merging in order to build the larger sorted array.

The conclusion from the above is that **quicksort** is **competitive** with **mergesort** and other **efficient sorting methods**. Both **quicksort** and **mergesort** are readily available in all common programming languages via standard libraries, you can use them for any sorting tasks you may encounter while solving interesting problems, and now you understand how they work. Both of them are **recursive**, and the central difference between the algorithms is **how the recursive case is handled**. We have seen that the choice of **recursive case** matters greatly, and can mean the difference between an **efficient** algorithm, and a **slow, impractical** one.

We have also gone through the process of designing a **recursive solution** for a general problem: **Sorting**. This should help you when you find a need to approach other problems with **recursion**. Next, we should spend some

time considering the performance implications of **recursive algorithms**, as they deserve careful thought and should not be ignored when considering whether or not **recursion** is the right tool for a specific task or problem.

5.6 Implementation considerations for recursive methods

Recursion requires a function to call itself a (possibly large) number of times. Recall, from the memory model in Chapter 2, that every time call a function is called, **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 is completed, and it returns a value, the space reserved for the function is released.

However, with recursive code this can become a problem unless we are careful - at any given time, there will be multiple (and possibly many) **active calls** to the recursive function. These are function calls that are **waiting from a result** from another recursive call, and until they receive that result, they hang around in memory and occupy space.

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 what happens in memory with recursive function calls).

The recursive function that computes the sum of the entries in an array works as follows (pseudo code):

```
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.

The input parameter **n** is used to indicate the element of the array that the function is adding at a specific point in the process - it is intended to start at the end of the array and work backward toward the first element in the array. The **recursive case** simply decreases **n** by one, which has the effect of reducing the size of the array at each step. Once the **base case** is reached, the recursion rebuilds the sum from the first element back to the last.

The recursive process for a small input array would look as follows:

```
sum([10,5,3,2,1,8],5)                    // First call, n=5, not base case      (A)
    sum([10,5,3,2,1,8],4)                // recursive call with n=4, not base case (B)
        sum([10,5,3,2,1,8],3)            // recursive call with n=3, not base case (C)
            sum([10,5,3,2,1,8),2)        // recursive call with n=2, not base case (D)
                sum([10,5,3,2,1,8],1)    // recursive call with n=1, not base case (E)
                    sum([10,5,3,2,1,8],0) // recursive call with n=0, base case! (F)

                    // At this point there are 6 active calls to 'sum()', each with
                    // their own input parameters, local variables, and space for
                    // a return value! - these correspond to (A) - (F)

                    // The last call can be completed, so (F) returns 10 to (E)
                    // space for (F) is released

                (E) - can now be completed, returns 5+10, space for (E) is released
            (D) - can now be completed, returns 3+15, space for (D) is released
        (C) - can now be completed, returns 2+18, space for (C) is released
    (B) - can now be completed, returns 1+20, space for (B) is released
(A) - finally, the original call can be completed, returns 29, and space for (A) is
    released
```

The same process is illustrated in Fig. 5.19 where you can see how memory is reserved for each function tall inside a **call stack** - this is a region of the computer memory that is set aside for the function calls. Any memory requested by any of the functions in the program is reserved in the **stack**, and it is reserved directly **on top** of any memory being used by other **active calls**. The specific region of memory assigned to each function call is referred to as the function's **stack frame**. In Fig. 5.19, each block (shown as a rectangle) for one function call represents a separate **stack frame**.

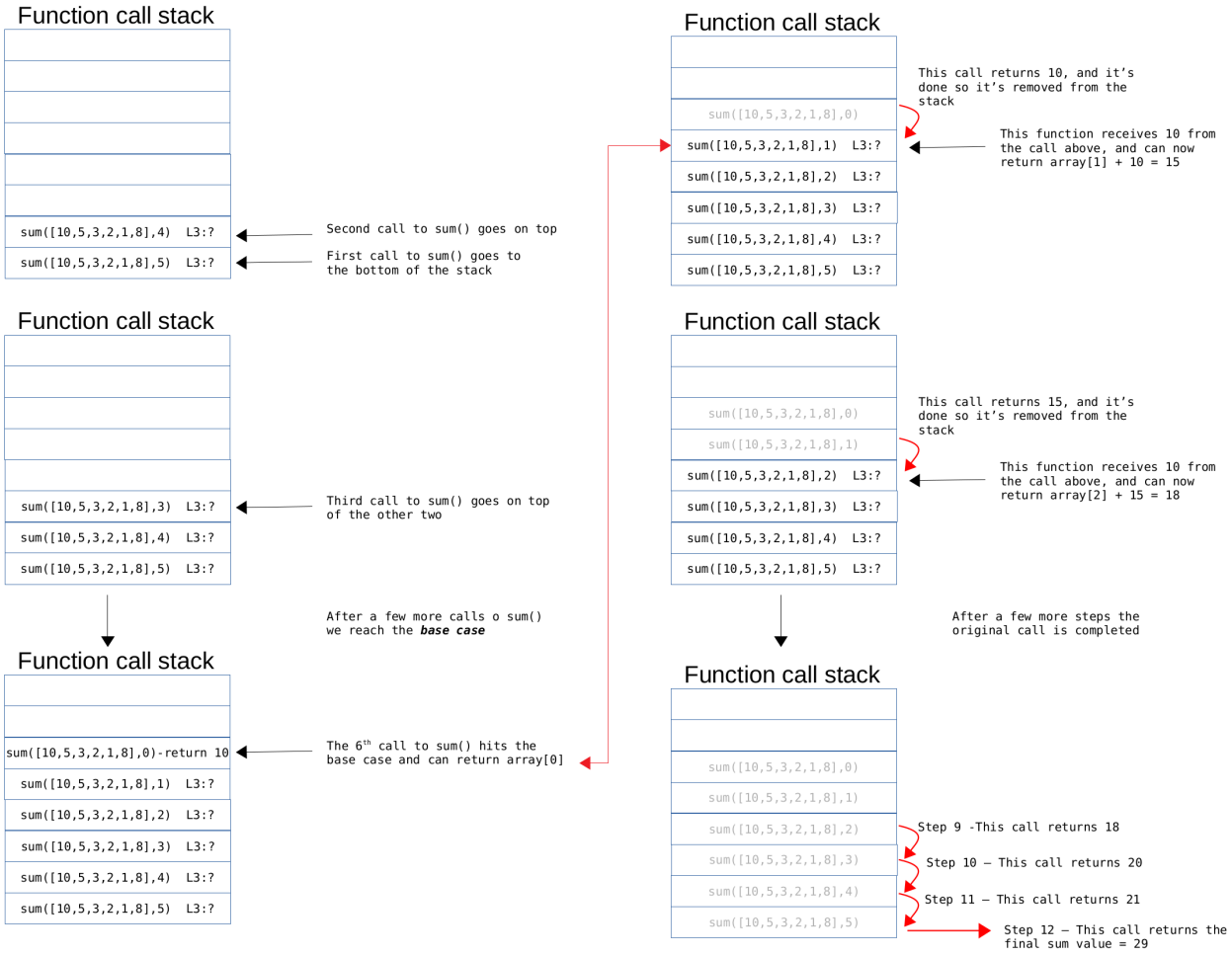


Figure 5.19: Illustration of how memory space is reserved in the **function call stack** for the recursive function calls to **sum()**. Each successive call goes to the **top of the stack**, previous calls remain **active** - they are waiting for a result and can not finish their work until the recursion returns the values they need.

There are two important implications from the way the process shown above develops:

- Recursive function calls take up space. Depending on the **depth of the recursion**, which is the number of calls required until the **base case** is reached, the space required may be significant. For an array of size N, the **sum()** function above will require N separate chunks of space in the **call stack**.
- Reserving space in the **function call stack** and releasing it when function calls return requires **work**. It is done automatically, but the computer still takes some small amount of time to reserve space and then to

release it when no longer needed. Once again, this may become significant if the **recursion depth** is large.

These two factors have important implications for **recursive functions**: They require significantly more memory than **non-recursive** functions, and in fact if we are not careful we run the risk of running **out of memory in the call stack** which causes a program to crash. Secondly, they are **slower** than **non-recursive** functions because of the extra work required to reserve and later release memory from the **call stack**.

When used for problems that have a **recursive nature** (as explained in detail in Section 5.5), the additional memory requirements and small overhead of handling memory in the **call stack** are balanced by the simplicity, elegance, and intuitive nature of the **recursive solution** - in such cases, a **non-recursive** solution is likely to be much more involved, harder to understand, maintain, test, and debug; and likely will require its own additional memory in order to keep track of information that the **recursive** solution automatically maintains via the **call stack**.

However, for certain problems that could equally well be solved with and without **recursion**, the choice of whether or not to implement a **recursive** algorithm should have carefully considered the extra space and slower nature of the recursive process.

Note

It is important to remember that for every problem we can solve with a computer, it is possible to find a **recursive** solution, as well as a **non-recursive** solution. Though for many cases it may not be obvious how to develop one or the other. There is no in-principle limitation to what kind of problems can be solved **recursively**, and the same is true for **iterative** (non-recursive) methods.

5.6.1 Tail-recursion optimization

The extra memory required for **recursion**, and the overhead of managing the **function call stack** can sometimes be avoided by using a technique called **tail-recursion optimization**.

In the example above, the **sum()** function builds 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 result.

This line of pseudo code 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 a result. 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). This forces us to keep the function's **stack frame** in the **stack** until the function's work is done.

However, we could achieve the same result in a slightly different way. Instead of building the sum backward from each successive recursive call, we could **pass the partial sums forward** into the recursion - each successive call to the **recursive function** does all of its work **before the recursive call**, and thus it no longer needs to hang around in the **call stack**.

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: It takes one extra parameter, **the partial sum** computed by previous calls to **sum_TR()**. The recursive case in **L3** is a bit different as well:

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

In this version, there is **no computation** and **no work** done **after the recursive call** - once we call **sum_TR()** recursively, we are done and whatever the recursive call returns, that is the result we want. Because the **recursive call** is the **last thing the function does**, we call this a **tail-recursive** function.

Let's see how this may change the process when the **tail-recursive** version of the sum function is called with the same sample array as the original, recursive **sum()** function discussed earlier:

```
sum_TR([10,5,3,2,1,8],5,0)                  // First call, n=5, part_sum=0  (A)

// The call in (A) passes a partial result, array[5]+0 = 8 into the recursive call.
// it's work is done so we do not need to keep it in the stack! memory for (A) is released

sum_TR([10,5,3,2,1,8],4,8)                  // Second call, n=4, part_sum=8 (B)

// The call in (B) passes a partial result, array[4]+8 = 9 into the recursive call.
// (B) is done, we can release its memory from the stack

sum_TR([10,5,3,2,1,8],3,9)                  // Third call, n=3, part_sum=9  (C)

// (C) passes a partial result, array[3]+9 = 11 into the recursive call.
// (C) is done, we release its memory from the stack

sum_TR([10,5,3,2,1,8],2,11)                 // 4th call, n=2, part_sum=11   (D)

// (D) passes a partial result, array[2]+11 = 14 into the recursive call
// (D) is done, removed from the stack

sum_TR([10,5,3,2,1,8],1,14)                 // 5th call, n=1, part_sum=11   (E)

// (E) passes a partial result, array[1]+14 = 19 into the recursive call
// (E) is done, removed from the stack

sum_TR([10,5,3,2,1,8],0,19)                 // 6th call, n=0, part_sum=19   (F)

// (F) reaches the base case, and returns array[0]+19 = 29, which is the final
// result. (F) is done and removed from the stack, the final result is returned.
```

The process above is illustrated in Fig. 5.20.

With a small change to how our **recursive function** is implemented, we **eliminate** the need to keep multiple **stack frames** for the function within the **call stack**. This means the **tail-recursive** version can handle inputs that are much bigger (i.e. it works for much larger N) than the original **non-tail-recursive** version. Secondly, the result of the computation is available **as soon as we hit the base case**, no additional work is needed. The result of these two factors means that the **tail-recursive** version of the sum function is **as efficient** as a **iterative** (non-recursive, using for loops) function that computes the sum of the entries in an array.

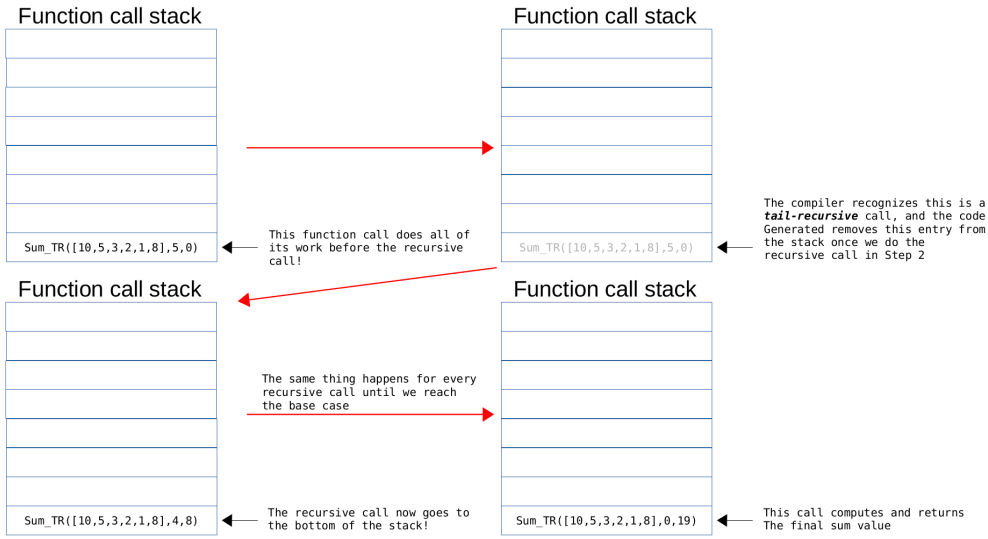


Figure 5.20: Illustration of how the **tail-recursive** function works at the **function call stack** level. Because each call **completes its work** before the recursive call takes place, it does not need to be kept in the stack. As a result there is never more than **one stack frame** reserved for the function in the **stack**, and in fact **the same stack frame** can be re-used by each successive call - thereby eliminating the **overhead** of **reserving and releasing space** from the **call stack**.

To test this, we can measure the time it takes to compute the sum of the entries in an array with $10,00$ floating point numbers using three different implementations of the sum function:

- **Iterative** sum - with for loops
- **Recursive** sum - the original **sum()**
- **Tail-recursive** sum - the **sum_TR()** we just developed

the sum is performed $100,000$ times with each function, and the total time is reported (doing the sum a single time is just too fast to accurately measure time taken on modern computers). The results are as follows:

```
./sum_runtime_analysis
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
```

We can quickly see that the **plain recursive** version of the sum function is about **70%** slower than the **iterative** function using only for-loops. We can also see that the **tail-recursive** version is **just as fast** - there is no visible **overhead** caused by the use of **recursion** (the tiny difference is not meaningful, and likely caused by timing inaccuracies in the computer where the program was run).

Note

Tail recursion optimization is a feature of modern programming languages and modern compilers. The compiler, or the interpreter for the language, must be able to recognize that a function is **tail-recursive**, and must be able to carry out the required optimizations to ensure no space is reserved in the **call stack** for successive calls to the **recursive function**, and that the result is returned directly from the last call when it reaches the **base case**. Put in a different way, if the compiler or language does not support **tail recursion optimization**, the two functions **sum()** and **sum_TR()** will behave in exactly the same way, use up a lot of space in the **stack**, and both would be just as slow when compared to the **iterative** version. Luckily, most current compilers and programming languages support this optimization.

So, in conclusion, when you are thinking of solving a problem with **recursion**, it is well worth the time to think of whether or not you can design your solution so that it is **tail-recursive**. It should be noted that **not all recursive functions can be made tail-recursive**. In such cases, the **overhead** of recursion must be carefully considered and accounted for.

5.7 Solving graph problems with recursion

We started this Chapter 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 shown in Fig. 5.21. In this problem, we have a **start location** (the mouse), a **destination** (the cheese), and a **graph** that has $8 \times 8 = 64$ different locations. Each of these is represented by a **node**. To identify which **node** corresponds to **which** location, we can number the nodes starting with 0 at the **top-left** corner of the maze, and going all the way to 63 at the **bottom-right**.

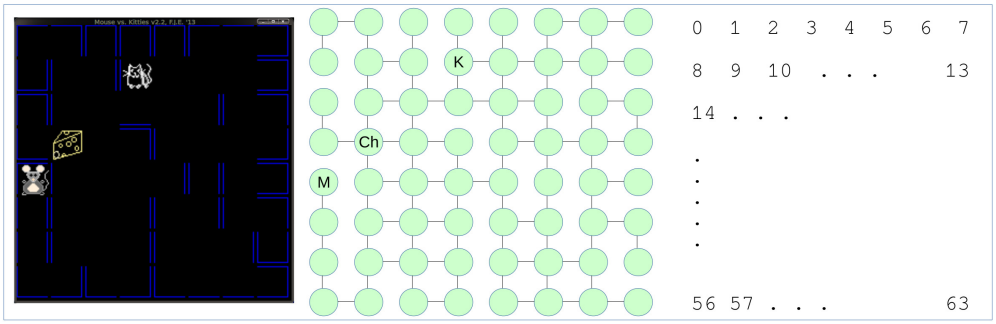


Figure 5.21: The problem of finding a path in a maze. Each location in the maze becomes a **node** in a **graph**, the **index** for the node corresponds to its **location** in the maze as shown.

We can represent the **connectivity** of the maze with the **edges** in the graph. If **two neighbouring locations** are **connected** (i.e. if the mouse can move from one to the next, because there is no wall separating them) we will have an **edge** joining the corresponding **nodes**. **Edge** information can be stored in an **adjacency matrix** or an **adjacency list**. For simplicity, here we will assume an **adjacency matrix** is used.

The problem we have to solve is finding a path, which consists of a sequence of connected nodes, leading from the start node to the destination. Let's formulate a **recursive** solution for this problem.

The base case: The **simplest** version of this problem occurs when **the start node** is the same as the **destination** node, in which case we are done. This is analogous to how a single-entry array is already sorted and there is nothing more to do. This is, however, not the only **base case** - what would happen if there was no path from **start** to **destination**? in that case we would expect to search through the entire graph and come up with no solution. So another **base case** is: there are no more **nodes** in the graph that we can **reach** from the **start node**.

The recursive case: This will deal with finding a path from **some starting** node (let's call it the **current node**) to the **destination**. It must reduce the size of the problem in some way so that eventually we reach one of our **base cases**. In the case of path finding, a common **recursive case**:

- Removes the **current node** from the graph (to create a smaller sub-graph).
- Recursively finds a path from **each neighbour of the current node** to the **destination** in the smaller sub-graph.
- If any of the neighbours returns a valid path, **build** a longer path from the **current node** to the **destination** by **pre-pending** the **current node** to the path returned by the neighbour.

Implicit in the above is that the **recursive case** will consider each **node** in the **graph** at most once (once it is removed by the **recursive case**, it will never again be reached from any of the sub-graphs).

The **order** in which we check for a path from the neighbours to the **destination** is important, it is called the **exploration strategy**. Different **strategies** will result in different search patterns (much like different ways for splitting an array result in different sorting methods). Depending on our choice of **strategy**, we may find completely different paths from one node to another. The choice of ordering strategy is a topic for an Artificial Intelligence book. There is no strategy that is **optimal for every graph search problem**, instead, different problems require different strategies.

For the purpose of understanding how we can apply **recursion** to solve path finding on **graphs**, let's look at the pseudo-code for the simplest strategy - one that is **purely recursive**. 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 **constraint satisfaction** problems.

```
// Inputs: current             (index for the current node)
//         destination         (index for the destination node)
//         adjacency_matrix[][] (NxN array representing edges in the graph)
//         visited[]           (an array of size N, initially all zeros)

DFS(current, destination, adjacency_matrix, visited)

    visited[current]=1;          // Mark current node as 'visited' (this
                                 // removes it from the subgraph)

  if 'current' is the same as 'destination'

       return destination        // We got to our first base case

  otherwise

    for each neighbour of current, if visited[neighbour]==0

      subpath=DFS(neighbour, destination, adjacency_matrix, visited)
```

```
    if 'subpath' is not <empty>

      // prepend current to subpath to build full path
            return {current -> subpath}

      // We checked all neighbours and found no path...

  return (<empty>) // Second base case, no subpath from 'current'
                   // to 'destination' can be found
```

The **DFS** algorithm as shown above is well worth knowing. If is found in many application domains, and can be used to solve many general problems that require exploring a **graph**.

✍ **Exercise 5.7** Implement **DFS** for path finding on a grid of locations just as in Fig. 5.21. 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 $m \times n$ the adjacency matrix has a size of $mn \times mn$) and then implement the **DFS** algorithm above.

Here is a sample **adjacency matrix** you can use to develop and test your **DFS** implementation. This is for a 4×4 grid:

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

Once you have completed the implementation of **DFS**, test it by finding paths between different pairs of nodes in the graph and checking that they make sense for the grid we have. This will require you to:

- Draw the graph as a grid of 4×4 nodes, linked as per the **adjacency matrix**
- Follow the paths returned by **DFS** for each of your tests, and verify they are valid (i.e. they visit neighbouring nodes that are connected in a sequence that leads from **start** to **destination**)

Note: The **DFS** process includes a step where the path is grown by pre-pending 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.

5.8 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 we need to understand before we start debugging:

- The base cases - double-check that every base case has been considered, and that they are all in the implementation.
- The recursive case - check that the the splitting strategy is reasonable, and that it is implemented properly. Write down a step-by-step example of the entire process for an input that is small (so you can walk through the entire process by hand) but large enough to get the recursion going for a couple of levels at least.

Tracing your recursive code: The tricky part here is that **the same function gets called over-and-over-and-over**. So we **need additional information** in order to figure out what step of the **recursion** each call corresponds to, and then determine at which step something is not working right. How can we do this?

First find a small-enough case that fails: As we saw at the end of Chapter 4, **debugging** starts with a failed test. So it is important to find an input that is **small enough you can verify the process by hand** and for which the program produces an incorrect result. Once we have the test case as well as a hand-written step-by-step solution for that test case, choose whether we want to use a **debugger** or whether we want to use **print statements** to trace through the program.

Tracing using a debugger: If we are using a **debugger**, then we want to **insert breakpoints** at the **start of the recursive function**, so that once the recursive function begins we can **step through** the code in the function and compare what it is doing against our **hand-written solution** for the selected test case. The goal is to verify that at each step the function carries out the correct process, that the **recursive case** is correctly splitting the problem into sub-problems, and that the **base cases** are being triggered when found. We will need to have pen and paper, as we will need to keep track of **recursion depth** and possibly other information needed to indicate what step of the process we are looking at.

Tracing using printf(): The goal is the same as when we are using a **debugger**, only in this case it is not interactive. Information is printed out for us to examine after the program has run. Because the **recursion** involves a sequence of calls, there will likely be a sizable number of lines of text with information for us to walk through and compare with out hand-written solution. To make the process easier we can do **one or more** of the following:

- We can add a parameter to your function that indicates the **recursion depth** - that is, the step within the recursion to which any printed information for this particular call corresponds. The **recursion depth** starts at **zero** when we first call the recursive function, and thereafter is increased by **1** by each successive recursive call.
- We can **indent** the information printed based on the **recursion depth** - so there is a visual indication of what level within the recursion the printed data corresponds to.
- We can **dump the printed information to a log file** - because reading from the screen makes it harder to go back and forth as we are looking for a problem, and it is possible that if there is a large number of print

statements, the earlier ones may have been lost because the terminal keeps only a fixed number of lines of text.

- We can add a **pause** within the recursive function (at an appropriate place) to give us time to check the information that was printed out for each **recursive** call before continuing with the next one.

Note

Dumping the program's output into a log file is easily done:

```
// On Linux/Mac (from the terminal)
./my_program  >  log.txt

// On Windows
.\my_program.exe > log.txt

// The above will result in a new text file called 'log.txt'
// that contains the output of any printf() statements in
// 'my_program'
```

Adding a **pause** statement is easily accomplished by adding something like this:

```
printf("Press [ENTER] to continue...\n");
fgets(&dummy[0], 10, stdin);

// you must have declared a dummy string somewhere else in the
// function:
// char dummy[10];
```

this simply waits for the user to input some string. For the purpose of debugging, it waits for you to press **ENTER**. Notice that any text that is typed-in will be ignored.

Having a well organized **log** can make the difference between being able to quickly locate a problem, or spending a large amount of time trying to make sense of the information printed out in the log. The closer the structure of the printed output resembles the structure of the recursion (illustrated in Fig. 5.22) the easier it will be for us to figure out where any problems may be.

A couple common problems often found with code that uses recursion:

- It never reaches the **base case** (we can see the **depth** increasing to unreasonable values given the input) - we should check the **recursive case**, make sure it is making the problem smaller, and then check that the **base cases** have all been identified and implemented.
- It runs out of **stack space** for **recursive** calls. This can happen with large problems which require very deep sequences of recursive calls and use a large amount of memory for each call. Consider whether it would be possible to implement a **tail recursive** version of the function. It this is not feasible, we can use the **depth** variable to enforce a **maximum allowed recursion depth**. Our **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 big to solve with the available stack memory. The advantage of doing this is that it prevents the program from crashing.
- The **recursion** doesn't return the correct solution - we should check that the **base case** is returning the correct

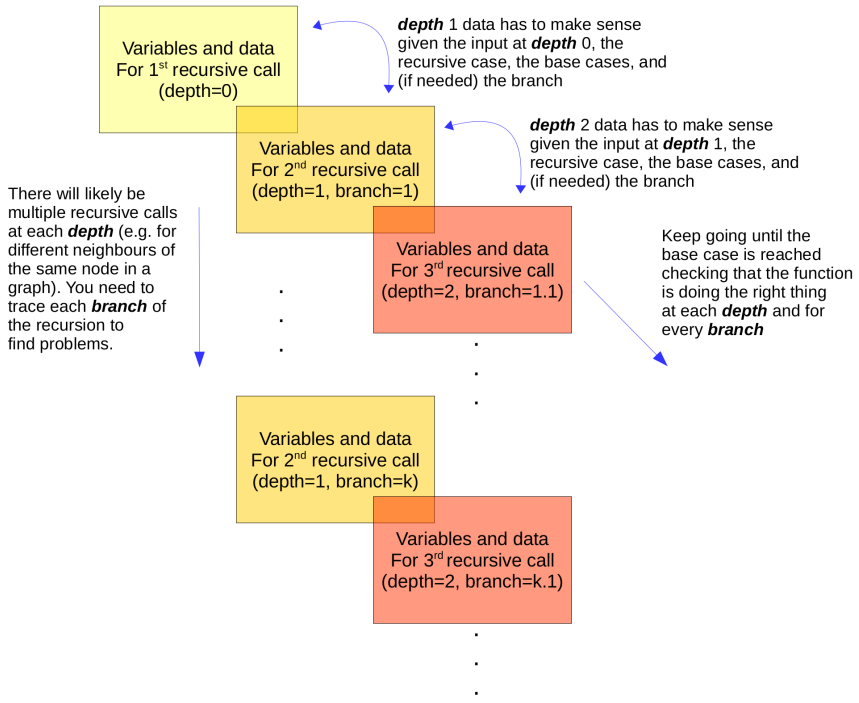


Figure 5.22: Structure of the **recursive process** showing how data at different **depths** relates to previous recursive calls. Whatever tracing process we choose to use, we need to remain aware of what step in the recursion we are looking at, so we can check the information available to us against our hand-written solution.

result, and then check for correctness the process of building a bigger solution from the smaller one.

This concludes our study of **recursion**, one of the most important and useful tools for problem solving in computer science. It is a concept that is often in the minds of people who work with computers, so we close this Section with a couple of cartoons about recursion, courtesy of Randall Munroe of **XKCD** (`http://www.xkcd.com`).

5.9 Additional Exercises

The importance of choosing the right tool - We have claimed that you can solve any problem 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.

✍ **Exercise 5.8** Assume we have a **BST** that stores unique **integer keys**. Write an algorithm that performs **in-order BST traversal** on this **BST** and prints 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, you will need to do some book-keeping that a recursive solution implicitly takes care of).

✍ **Exercise 5.9 Flood-fill:** A common operation in paint programs is that of filling a closed region with colour. This is called **flood-fill** because we can think of it as pouring paint into the region, which then spreads out until it reaches the boundary of the region. The process is illustrated in Fig. 5.24.

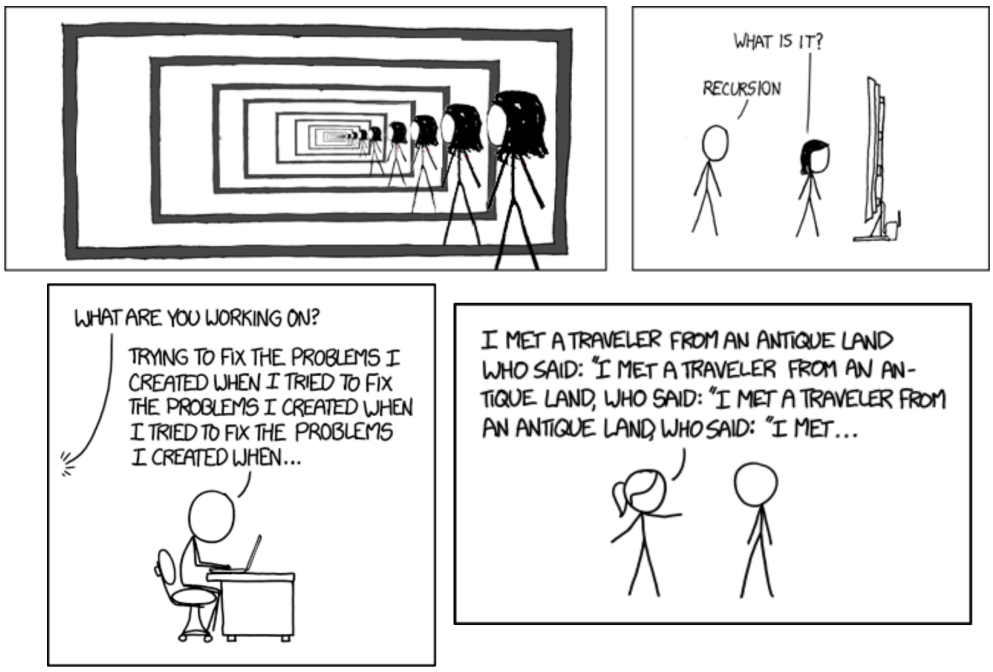


Figure 5.23: A couple of cartoons about **recursion** from **XKCD**. *Courtesy of Randall Munroe, xkcd.com.*

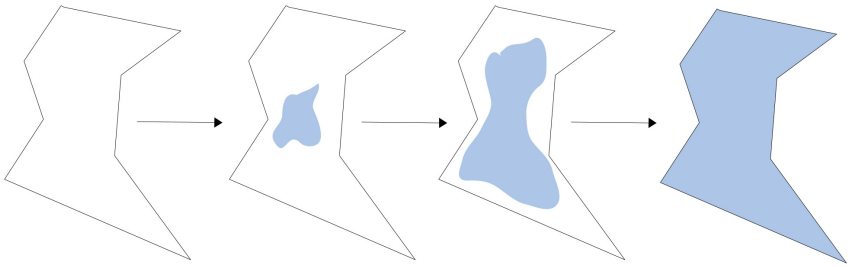


Figure 5.24: The **flood-fill** process simulates pouring paint into a closed shape. The paint spreads out until it reaches the boundaries of the shape. The process stops when the entire shape is filled with paint.

For the purpose of this exercise, the image will be a 2D array of size 500×500, and it will represent a **black and white** picture. Black pixels will have a value of **0** and white pixels will have a value of **1**.

- Write the algorithm for performing **flood-fill** on a closed shape such as shown in Fig. 5.24 using only loops, no recursion
- Write the algorithm for performing **flood-fill** using **recursion**.

Hint: For both cases, it helps to think of the process of paint spreading in order to figure out what the algorithm should do next.

✍ **Exercise 5.10 Graph distance** - One common problem in **graph-based applications** is finding the subset of nodes that is reachable from a given **node** within a certain number of **hops** (i.e. going from one **node** to a neighbouring **node** is one **hop**). For example, suppose we have a **graph** representing the pre-requisite structure of courses at a University, and we want to know which courses have **Introduction to Cell Biology** as a pre-requisite, or as a pre-requisite to their pre-requisite. This would require us to find any **nodes** in the **pre-requisite graph** that are up to two hops away from **Introduction to Cell Biology**. Another example, in a **social network** representing people (as **nodes**) and their social connections **as edges**, we may ask for a list of people who are either **friends**, **friends-of-friends**, or **friends-of-friends-of-friends** of a particular person in order to invite everyone to a big party. This is equivalent to finding all the **nodes** in the social network **graph** that are at a distance of **3 or less** from the one representing the person whose friends we want to invite over.

The task: Given the adjacency matrix for the graph:

- Propose an algorithm that finds all **nodes** within a distance **k or fewer hops** from a selected **start node** using only loops.
- Propose an equivalent algorithm (i.e. one that solves exactly the same task) using **recursion**.

Question: Which of the two versions of the algorithm is more **intuitive** and **easier to understand and generalize**?

✍ **Exercise 5.11 Crunchy!** - **DFS** is only one possible **strategy** for exploring a **graph**. A different method, called **breadth-first search (BFS)** uses a different order of exploration. For **DFS**, a node's grand-children, great-grand-children, and so on, will be explored before all of the node's children are explored. To see this, trace through the **DFS** pseudo code on a small graph (say, 4×4) and notice the order in which **nodes** are visited.

With **BFS** on the other hand, all the node's children are all explored first (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**). Develop **pseudo code** for implementing the **BFS** algorithm for exploring a graph. Assume you have an implementation of a **queue** (you may want to review the operations supported by a **queue** in Chapter 3).

BFS is used for graph search, so it may help to think in terms of the path-finding problem described earlier in the chapter. You can assume there will be a **start node** and a **destination node**, and we want **BFS** to find and return a path from **start** to **destination**.

✍ **Exercise 5.12 Tail recursion:** - Implement a function that computes the **factorial** of an integer n **iteratively** (using only loops). Then implement a **recursive** version of the function. Finally, see if you can implement the **recursive** version so that it is **tail-recursive**.

Once you have the three different implementations, test them for **speed** by

- Writing a pair of **nested for loops**. The outer loop runs **100 iterations**, the inner loop runs **1,000,000** iterations. In the body of the inner loop there is a single instruction that computes the factorial of some number, e.g. 25, using **one** the three functions you wrote
- Compile and run the program, and time how long it takes to complete
- Change the function used to compute the factorial, and run the program again. Record the run-time for all three different implementations

Note: To measure the time it takes for a program to complete you can do the following on Linux and Mac:

```
>time my_program
```

this will print out the time taken by **my_program**

✍ **Exercise 5.13 Tail recursion:** - Implement a function that computes the n^{th} **Fibonacci** number using only loops. Then implement a **recursive** version of the function. Finally, implement a **tail-recursive** version of the function. Once you have the three different implementations, test them for **speed** as described in the previous exercise.

5.10 Building programs that work - Part 5

Through the previous chapters, we developed a process for **solving a problem**, **developing our solution into a program design**, **implementing and testing as we develop** to produce code that works, and **debugging** to find and resolve problems identified through testing or that show up once users are running our programs.

Now we should look at two very powerful tools to help us find and fix a wide variety of problems that result from bugs in how our program **uses or accesses memory**. This class of bugs tends to be particularly tricky to find and fix using standard tools such as **debuggers** or **print statements**. This is because often their behaviour is somewhat random, and may or may not be **triggered** by a specific test, or may occur under particular testing conditions which are hard to identify.

Luckily, there are tools designed to **check every memory access** our program does, and report any problems so we can fix them. These tools are:

- **valgrind**, a free **open-source** tool that has been the standard for memory checking and debugging on Linux and Mac for a long time. This can be installed directly from your computer's **package manager**.
- **Dr. Memory**, a newer, free **open-source** tool that is available for all platforms: Linux, Mac, and Windows, you can find it here: https://drmemory.org/.

Both carry out a thorough check of your program's memory accesses, and can help you find a large variety of bugs including very common ones such as:

- **Off-by-one errors** - and in general **index-out-of-bounds** errors when working with arrays.

- **Invalid memory access** - such as using a pointer and an offset to read or write into memory that is not reserved for the program.
- **Use of uninitialized variables** - which happens when we declare a variable, and then use it **before** we actually assign a value to it. As you remember, variables contain **junk** until we give them a value.
- **Use of uninitialized data** - similar to uninitialized variables, but happens when we use **malloc()** to get memory and then forget to fill that memory with valid data before using it in the program in some way.
- **Memory leaks** - which occur when we reserve memory with **malloc()** or **calloc()** but forget to release it with **free()** before the program ends.
- **Double free or invalid free** - which happens when we try to release memory more than once, or when we try to release memory that was not reserved with **malloc()** or **calloc()**.

Both of these tools offer a significant advantage when trying to resolve bugs related to memory access. But bear in mind that **both of these tools are complex** and they will require you to practice **using them** and **understanding the output**. There are extensive tutorials for both **valgrind** and **Dr. Memory**, and you should spend a bit of time learning how to use at least one of these.

But, to give you an idea of what these tools do, here is a short program with plenty of memory access problems:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int array[10];
  int d;
  int *p;
  float *fp;

  // Fill the array with values
  // depending on if d=0 or
  // d=1 (unfortunately, it seems we
  // forgot to set d to some value!)
  if (d==0)                                  // Problem #0
  {
    for (int i=0; i<10; i++)
      array[i]=i;
  }
  else
  {
    for (int i=0; i<10; i++)
      array[i]=i*i;
  }

  // Print the array, unfortunately
  // we have an off-by-one error
  // in the for loop!
  for (int i=0; i<=10; i++)                  // Problem #1
   printf("array[%d]=%d\n",i,array[i]);

  // Get a pointer to the array and use
  // it to change some values,
  // unfortunately we are trying to access
```

```
  // values that are not in the array
  p=&array[0];
  *(p+250)=15;                                  // Problem #2

  // Let's get some memory for floating point
  // data
  fp=(float *)malloc(5*sizeof(float)); // 5 floats requested

  // And the line below should crash the program
  // because we are trying to free memory for
  // an array that was not dynamically allocated!

  free(p);                                      // Problem #3

  return 0;  // All good?

  // No! we forgot to free() the memory assigned to
  // fp, so there is a memory leak!

                                                // Problem #4
}
```

See what happens when we compile and run the program:

```
>./a.out
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=1955339008
double free or corruption (out)
Aborted (core dumped)
```

Obviously something went wrong, so we need to **trace and debug** the program - in the example above it is easy because it is short and we already know what the problems are, but you can imagine in a large, complex piece of software finding a memory access issue will be difficult. Let's see what we can learn by running **Dr. Memory** and **valgrind** on the program above. The output from **Dr. Memory** is shown below:

```
> ./drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.6.0
~~Dr.M~~ WARNING: application is missing line number information.
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffeffd5495c-0x00007ffeffd54960 4 byte(s)
~~Dr.M~~ # 0 main
~~Dr.M~~ Note: @0:00:00.297 in thread 5316
~~Dr.M~~ Note: instruction: cmp 0xffffffbc(%rbp) $0x00000000
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
```

```
array[7]=7
array[8]=8
array[9]=9
array[10]=-1084233984
~~Dr.M~~
~~Dr.M~~ Error #2: INVALID HEAP ARGUMENT to free 0x00007ffeffd54970
~~Dr.M~~ # 0 replace_free           [/home/runner/work/drmemory/drmemory/common/alloc_replace.c
    :2710]
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.324 in thread 5316
~~Dr.M~~
~~Dr.M~~ Error #3: LEAK 20 direct bytes 0x00007f8cec2f13a0-0x00007f8cec2f13b4 + 0 indirect
    bytes
~~Dr.M~~ # 0 replace_malloc           [/home/runner/work/drmemory/drmemory/common/alloc_replace
    .c:2580]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~      0 unique,    0 total unaddressable access(es)
~~Dr.M~~      1 unique,    1 total uninitialized access(es)
~~Dr.M~~      1 unique,    1 total invalid heap argument(s)
~~Dr.M~~      0 unique,    0 total warning(s)
~~Dr.M~~      1 unique,    1 total,    20 byte(s) of leak(s)
~~Dr.M~~      0 unique,    0 total,     0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~     14 unique,   17 total,  7610 byte(s) of still-reachable allocation(s)
~~Dr.M~~         (re-run with "-show_reachable" for details)
~~Dr.M~~ Details: /home/strider/mtvp/DrMemory-Linux-2.6.0/drmemory/logs/DrMemory-a.out
    .5316.000/results.txt
```

Notice the line

```
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffeffd5495c-0x00007ffeffd54960 4 byte(s)
```

It indicates we are accessing **uninitialized memory** in the program. **it is not clear exactly where** this happens because **Dr. Memory** is working on the **executable program** and can't reference your program's code.

In order to help understand what memory debugging tools provide us, **it helps to use print statements to tell us which part of the program is running**. We are limited to using print statements because we **can not run a debugger together with a memory checking tool**. So - let us change our program and add a few print statements:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
  int array[10];
  int d;
  int *p;
  float *fp;

  // Fill the array with values
  // depending on if d=0 or
  // d=1 (unfortunately, it seems we
  // forgot to set d to some value!)

  printf("Initializing integer array... \n");
  if (d==0)                                  // Problem #0
```

```
  {
    for (int i=0; i<10; i++)
      array[i]=i;
  }
  else
  {
    for (int i=0; i<10; i++)
      array[i]=i*i;
  }

  // Print the array, unfortunately
  // we have an off-by-one error
  // in the for loop!

  printf("Printing the contents of the integer array... \n");
  for (int i=0; i<=10; i++)                 // Problem #1
   printf("array[%d]=%d\n",i,array[i]);

  // Get a pointer to the array and use
  // it to change some values,
  // unfortunately we are trying to acces
  // values that are not in the array
  printf("Using pointers to access array data... \n");
  p=&array[0];
  *(p+250)=15;                              // Problem #2

  // Let's get some memory for floating point
  // data
  printf("Reserving dynamic memory... \n");
  fp=(float *)malloc(5*sizeof(float)); // 5 floats requested

  // And the line below should crash the program
  // because we are trying to free memory for
  // an array that was not dynamically allocated!

  printf("Releasing memory before the program exits... \n");
  free(p);                                  // Problem #3

  return 0;  // All good?

  // No! we forgot to free() the memory assigned to
  // fp, so there is a memory leak!

                                            // Problem #4
}
```

Now let's re-compile the program and run **Dr. Memory** again, which results in the following output:

```
>./drmemory -- ./a.out
~~Dr.M~~ Dr. Memory version 2.6.0
Initializing integer array...
~~Dr.M~~ WARNING: application is missing line number information.
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 0x00007ffcce72498c-0x00007ffcce724990 4 byte(s)
~~Dr.M~~ # 0 main
~~Dr.M~~ Note: @0:00:00.272 in thread 5502
~~Dr.M~~ Note: instruction: cmp 0xfffffffbc(%rbp) $0x00000000
Printing the contents of the integer array...
array[0]=0
array[1]=1
```

```
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=724062720
Using pointers to access array data...
Reserving dynamic memory...
Releasing memory before the program exits...
~~Dr.M~~
~~Dr.M~~ Error #2: INVALID HEAP ARGUMENT to free 0x00007ffcce7249a0
~~Dr.M~~ # 0 replace_free           [/home/runner/work/drmemory/drmemory/common/alloc_replace.c
    :2710]
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.295 in thread 5502
~~Dr.M~~
~~Dr.M~~ Error #3: LEAK 20 direct bytes 0x00007f0c80e213a0-0x00007f0c80e213b4 + 0 indirect
    bytes
~~Dr.M~~ # 0 replace_malloc          [/home/runner/work/drmemory/drmemory/common/alloc_replace
    .c:2580]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~      0 unique,    0 total unaddressable access(es)
~~Dr.M~~      1 unique,    1 total uninitialized access(es)
~~Dr.M~~      1 unique,    1 total invalid heap argument(s)
~~Dr.M~~      0 unique,    0 total warning(s)
~~Dr.M~~      1 unique,    1 total,    20 byte(s) of leak(s)
~~Dr.M~~      0 unique,    0 total,     0 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~     14 unique,   17 total,  7610 byte(s) of still-reachable allocation(s)
~~Dr.M~~         (re-run with "-show_reachable" for details)
~~Dr.M~~ Details: /home/strider/mtvp/DrMemory-Linux-2.6.0/drmemory/logs/DrMemory-a.out
    .5502.000/results.txt
```

Now we can see that **Error #1** occurred while **initializing integer array**, so it must be reporting the use of **d**, which was declared but not initialized (this is labeled **problem #0** in the program listing).

Interestingly, **Dr. Memory** did not catch the **off-by-one** error printing the array, or the **invalid pointer access** - this is likely because whatever memory locations were accessed are still within the program's reserved memory.

The next error to be reported, **Error #2** happens when **releasing memory before the program exits...**, and it states that an invalid argument was passed to **free()**. This corresponds to **problem #3** in the program listing.

Finally, **Dr. Memory** reports in **Error #3** that there were **20 bytes** of **memory leaks** (**problem #4** in the listing) corresponding to the 5 floating point values we allocated with **malloc()** and assigned to pointer **fp** but forgot to release.

All things considered, by using a combination of **carefully placed print statements** together with **Dr. Memory**, we were able to quickly identify 3 out of 5 problems in the program above. The remaining two would require **specific tests that cause the program to fail** to be run, together with the **print statement + memory checking** combination.

Let's see what **valgrind** does with the same program (**valgrind** prints a lot of information messages that are not relevant for debugging, the output below contains only the lines pertaining problems with the program):

```
> valgrind --verbose ./a.out
==5911== Memcheck, a memory error detector
==5911== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5911== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5911== Command: ./a.out
==5911==
--5911-- Valgrind options:
--5911--   --verbose

.
.
.

--5911-- REDIR: 0x4ed5020 (libc.so.6:malloc) redirected to 0x4c31aa0 (malloc)
Initializing integer array...
==5911== Conditional jump or move depends on uninitialised value(s)
==5911==   at 0x1087B1: main (in /home/strider/CS/course_material/MyCourses/Sabbatical_23-24/
    ICS_BOOK/scratch/a.out)
==5911==
--5911-- REDIR: 0x4fcc970 (libc.so.6:__mempcpy_avx_unaligned_erms) redirected to 0x4c39130 (
    mempcpy)
Printing the contents of the integer array...
--5911-- REDIR: 0x4fcc090 (libc.so.6:__strchrnul_avx2) redirected to 0x4c39020 (strchrnul)
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9
array[10]=1001336320
Using pointers to access array data...
Reserving dynamic memory...
Releasing memory before the program exits...
--5911-- REDIR: 0x4ed5910 (libc.so.6:free) redirected to 0x4c32cd0 (free)
==5911== Invalid free() / delete / delete[] / realloc()
==5911==   at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5911==   by 0x10888C: main (in /home/strider/CS/course_material/MyCourses/Sabbatical_23-24/
    ICS_BOOK/scratch/a.out)
==5911== Address 0x1ffefffb50 is on thread 1's stack
==5911== in frame #1, created by main (???:)

.
.
.

==5911==
==5911== LEAK SUMMARY:
==5911==   definitely lost: 20 bytes in 1 blocks
==5911==   indirectly lost: 0 bytes in 0 blocks
==5911==     possibly lost: 0 bytes in 0 blocks
==5911==   still reachable: 0 bytes in 0 blocks
==5911==        suppressed: 0 bytes in 0 blocks
==5911== Rerun with --leak-check=full to see details of leaked memory
```

```
==5911==
```

Similar to **Dr. Memory**, **valgrind** finds:

- Conditional jump or move depends on uninitalized value(s) - problem #0
- Invalid free() - problem #3
- Memory leaks 'definitely lost: 20 bytes in 1 block' - problem #4

Also just like **Dr. Memory**, **valgrind** didn't catch the **off-by-one** array access problem, or the **invalid pointer access**. However, any test case where your program crashes will be caught by either of these tools, and the possible cause will be reported.

In conclusion: You should learn to use, and then always check your programs with a **memory checking tool**. It will likely reveal problems that were not visible during your testing and that would go undetected otherwise. While the tools aren't perfect and will not catch every single problem with a program, used in conjunction with **a thorough and well designed testing strategy** they will enable you to produce code that is sound and correct in terms of how your programs access information within the computer's memory. Spending time learning how to use a **memory checking tool** will turn you into a much more capable software developer, and is well worth the effort. Another good habit to develop, and a valuable skill to add to your toolbox.