Algorithms and Data Structures

By Grady Fitzpatrick

Algorithms (continued)

Algorithmic Efficiency

Algorithmic efficiency is our answer to this problem; we pick some feature and work out how fast that feature expands as we increase the input size, for example how much space an algorithm uses for 10 items compared against how much it uses for 100 items. We also look at time efficiency with a similar method. We simply call the behaviour of the growth the "complexity" of the algorithm. Using these metrics, we typically use three measures:

Ω(f(n)), any algorithm, g(n) with complexity of Ω(f(n)) obeys the constraint

$$\mathrm{g(n)} \geq af(n)\ \forall\ \mathrm{n} > \mathrm{c}_1, where\ a\ and\ c_1\ constant$$

In other words, g(n) is more than some constant multiplied by f(n) when n passes a certain size, or g(n) grows faster than f(n). We call this measure "Big Ω of f(n)", and this typically is used in determining feasibility of difficult problems, it won't have as much usefulness in this course as the other two.

O(f(n)), any algorithm, g(n) with a complexity of O(f(n)) obeys the constraint

$$\mathrm{g(n)} \leq af(n)\ \forall\ n > c_1, where\ a\ and\ c_1\ constant$$

In other words, essentially f(n) is Ω(g(n)), g(n) grows slower than f(n) as the input size grows. In computing, we often use this as the "least upper bound", however the notation does not necessitate that, a function g(n) = 2n is O(n), but it is also O(n^2) and O(n^3), so be careful. We usually call this measure "Big O of f(n)"

And lastly, Θ(f(n)), any algorithm, g(n) with a complexity of Θ(f(n)) obeys the constraint

$$\mathrm{g(n)} \in O\big(f(n)\big) \wedge g(n) \in \Omega(g(n))$$

In other words, g(n) grows as fast as f(n) as the input size grows. This is also known as the "least upper bound", this is the most useful of the three as it tells you the most about how an algorithm will perform.

How do we determine g(n)?

Looking at the growth of the algorithms looked like it completely solved our problems until we actually try to make the function g(n), "n" refers to the size of the input we're sending into the algorithm, so for example, if we need to do a for loop with n iterations, this would add an "n" multiplier for what's inside that loop into our equation. One issue we immediately run into is that we aren't sure what to count, and because we're using this as a tool to determine the faster running algorithm, we make the assumption that each of our basic operations (add, multiply, divide, assign, etc.) are "1 operation", which allows us to get somewhere, however, this mandates that we know how every line of code in our program, which isn't terribly useful for picking what's best before we've started. The solution to this is to simply look at the fastest growing parts of our algorithm and simply compare those, for example, an algorithm with O(2n + 1) operations is going to grow similarly to an algorithm with O(n) operations, so it's clear we can strip the multiplier and keep only the highest order term "n". With this in mind, we can just look for things that increase this highest term, for example, if an algorithm has only operations that don't change based on the size of the input, we see that it has O(1) performance, however, if we see something that iterates over the entire input, we might see this rise to O(n) (this might be through a loop, or through some searches), and a loop the size of the input happens each iteration, we might see this jump to O(n^2), this makes the job much faster as we no longer have to look at each individual operation, but only the expensive parts.

What's wrong with the simplified view?

Though simplifying the algorithm run-time complexity into classes makes decisions easier, it's not perfect, if you have two algorithms that both run with Θ(n) complexity, one might have 2n + 10 operations and the other might have 3n + 2 operations, which means that for large n, the first algorithm will run far faster than the second one, despite having the same complexity class, and on the other hand, the latter algorithm will run better for (very) small n, meaning it may possibly be better. More drastically, if you have an algorithm that runs in Θ(2^n), and another algorithm that runs in Θ(n^2), you might find that the first algorithm runs better even up to somewhat large n, and may be appropriate to use in cases where you know the size of your input will never reach above a certain size. Further, even when two algorithms run 10 operations each, some small differences on how long the processer takes to process those instructions might be present meaning that one is faster than the other.

Does that mean it's useless?

No, not at all, the complexity classes are very useful in making high level decisions about what algorithms to use. It is important to understand that the complexity classes are simply explaining growth behaviour in a way that does not need the code written before it can be investigated.

What common measures are there for algorithms?

Often you will find that algorithms include best case, average case and worst case run-time or space complexities, of these, the best case is the least useful, it is primarily used as a benchmark to see how good the algorithm can get if data is balanced to take advantage of it. The average case is the average time complexity it takes for the algorithm to run; this is typically most useful in writing software that runs fastest. The worst case is the maximum time complexity that the algorithm will take; this is primarily used in cases where a hard limit on the performance of a piece of code is necessary, for example in safety-critical systems. When we say an algorithm has a time complexity of Θ(f(n)), we are saying all cases run with that complexity.

Data Structures

What are Data Structures?

Data Structures refer to specific arrangements of data, and often how that data is used. If algorithms are like the method in a recipe, data structures are like the ingredients. Knowing and understanding specific data structures allows us to build programs that perform well and to solve recurring problems, we can also use specific data structures together to create hybrid data structures which perform better than either pure data structure for the particular problem.

Abstract Data Structures

Abstract data structures are data structures that typically refer to a data structure that fits some kind of behaviour. An example of an abstract data structure is a dictionary, the dictionary is typically defined as allowing the lookup of some value based on a key; for example, looking for the key "John Smith" might return a picture of the person that's stored in the program's database. Exactly how this dictionary is implemented is not specified, and this could be fulfilled a number of possible data structures.

Concrete Data Structures

Concrete data structures are data structures that have both the behaviour of interaction with the data structure and the structure defined specifically. An example of a concrete data structure is the linked list; the linked list is a list made of "nodes", these nodes have a pointer to the "next" node and sometimes the "previous node", as well as some kind of data stored, the linked list is usually accessed using a "head" pointer to the first node in the list. The data structure also includes algorithms explaining how the list is traversed or searched, items are added, items are deleted and how new lists are created, if the linked list is being used as a dictionary, usually the data in each node will have the key as well as the data for that key. In concrete data structures like this linked list, there are some parameters that can be varied; however, the data structure behaviour is completely specified.

Why does it matter?

Understanding the difference between abstract data structures and concrete data structures allows better separation of programs, in order to create more modular code by separating structural behaviour from concrete behaviour. Modular code can be reused much more easily and can be less prone to bugs.

Data Structures in C

It is worth spending a little time explaining the basic data structures we use in C, a completely in depth explanation of how aspects of numbers in C are stored (such as twos compliment or mantissa allocations and the like in floats) is a topic that while interesting, does not provide a lot of insight into the issues programmers face but don't notice. Most issues that new programmers face that cause them trouble relate to two main aspects of C, data types and the malloc/free suite of functions.

C is not a strongly typed language, what this means is that data is simply data and its structure gives it no unchangeable rules about the meaning of a set of bytes. Some implicit actions occur to make life easier for programmers; however it is worth examining exactly what is done.

Data Types

C has five main types, the sizes of these on a typical 64-bit machine today are given here, however, this could vary and in fact these values were smaller in the past.

Type	Description	Typical Size
int	Used for integers, uses twos compliment in order to allow negative numbers	4 bytes
unsigned int	Used for integers, no negative numbers	4 bytes
double	Used for decimal numbers, the name comes from “double precision float” using twice as much space as a regular float	8 bytes
char	Used for representation of text	1 byte
pointer	Used to point to an address in memory	8 bytes
void	Used to designate that the given data has no type	1 byte

It is worth noting that there exist larger and smaller versions of both int and double values; and that pointers can’t be assigned without some qualifying type of the underlying data (this underlying data may or may not be of the type specified, however it must be specified so that a dereference operation can take place on it).

Type Casting

As with many languages, C allows programmers to write programs that change one type to another, for example, an integer can be converted to a double or unsigned integer, or vice versa, many people who have used C have used this facility and noticed for example that if a = 5 and b = 2, a / b might give 2, but (double) a / (double) b gives 2.5, this is an example of type casting. Some type casts also change behaviour instead of output values, for example, in pointers, typecasting an (int *) pointer to a (char *) pointer will not change the value output from the pointer; however the dereferenced pointer will give a different type of value.

Data Type Fluidity

As mentioned earlier, C is not strictly typed; this means that you can access an integer as if it were a double and vice versa. It is highly worth noting that this is different from type casting, for example, consider what the full set of possible values that could be stored in an integer are and compare those to the full set of possible values that could be stored in a float (a float is the 4 byte version of a double), so it is clear that something atypical will happen when the value of the float passes the maximum allowed value of the integer, using the type casting, this ends up maxing out the number ignoring the twos compliment (that is, it converts to the largest magnitude negative value possible). The key difference between type casting and non-strict typing is that clearly the underlying value of the float is changing as the value is increased, so accessing it as an integer will result in significantly different values than when accessed as a float. Further examples and explanation of this idea requires greater explanation of pointers.

Program Runtime Structure

This will be a very brief section on how C programs are structured during their operation. Programs are usually broken into three main parts, the static code pages, the stack and the heap. The static code pages are basically where the way program actually works is kept, this might include strings as well as functions and the rest of the actual program, this part of the program is rarely modified and in many cases is read-only. The next part of the program is the stack; a data structure which we will cover later on in the subject, the stack is managed by the program and handles things such as the values of variables for each function and the allocation and freeing of these values. The last part of the program is the heap; the heap is the only part of the program which the programmer really handles explicitly, when you allocate memory (for example, with the malloc/realloc/calloc functions) you generally take it from the heap, the heap is also the only part where you are able to free memory you have allocated (for fairly clear reasons). This is by no means an in depth view of each of the components during runtime, however it gives you an overview that helps you understand why certain parts work and other parts don't.

Pointers

Pointers are a simple concept that many early programmers struggle significantly with, often without realising it. Pointers simply contain a value which represents the address to a particular point in memory.

Dereferencing

Dereferencing is a fairly simple affair, the dereference operator * is used to access the memory at the given location, the dereferenced pointer is given the type of the dereferenced pointer, for example, an (int *) variable when dereferenced gives an int value, and an (int **) variable when dereferenced will give an (int *) value. Worth noting is that this occurs in a lot of cases through other operators, for example, [0] will usually perform the same way as *, since for example a[0] is *(a + 0). Worth noting about pointers is that additions to addresses are done in units of the type, this just means that for example, if you have two variables declared as
*int **a;*
*int *b;*
the address (a + 1) will be 8 bytes after a in memory (assuming pointer size of 8 bytes) and the address (b + 1) will be 4 bytes after b in memory (assuming int size of 4 bytes).

Addresses of Variables

Pointers might seem like a whole lot of fun based on what you've heard so far, and some functions return pointers already, however sometimes you need to do something with a piece of code that's on the stack, or maybe even in the static part of your program. In that case, C provides a nice tool, the address operator &, this operator can be used to get the address of a particular variable or function, rather than its value. For example, a variable declared as:
int ex = 7;
can be modified by sending the address of the variable ex, using &ex, into a function and then dereferencing that pointer. This is important because if a variable is sent as a parameter into a function, it will copy the contents of that variable into the function's stack and then the function will be working on that particular part of its stack (so it's version of ex if you don't use &ex). However, it is worth noting that even using &ex, this value is still copied into the function's part of the stack, it's just that the value copied in is the address of the variable, so we can dereference the address and work on the original variable.

Function Pointers

Function pointers are fairly simple conceptually, rather than pointing to a variable, they point to a function; this comes in extremely handy in producing modular code, as various concrete data structures can be loaded into abstract data structures to provide the desired functionality. The main issue people run into with function pointers is that they simply aren't sure how to specify them, the Right-Left rule mentioned below is likely to be extremely helpful in ensuring what code you've written actually means what you think it does, an example of the function pointer is
*int (*exampleFunction)(double, double);*
which would be a pointer to a function returning a double, the function can be assigned by using the address operator & and the name of the function. Such as:
exampleFunction = &doubleCompareFunc;
where doubleCompareFunc is a function taking two doubles as arguments and returning an int. For note, this assignment can be rolled into the declaration as well.

Composite Data Types

C provides a number of data types that are made up of other data types. The most commonly used of these in this subject is the struct.

Structs

The struct data type is a structuring technique that allows multiple "member elements" to be stored in a single block of data. A new stuct is created using the keyword struct. An example of a struct would be:

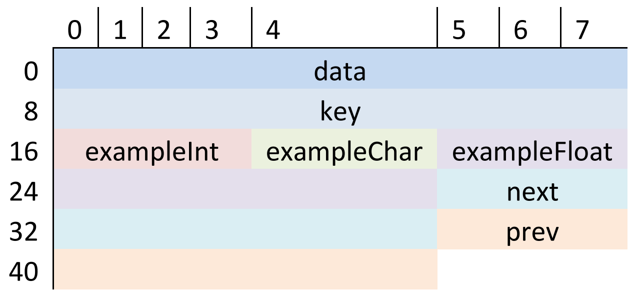
```
struct linkedList {
   void *data;
   void *key;
   int exampleInt;
   char exampleChar;
```

```
    double exampleFloat;
    struct linkedList *next;
    struct linkedList *prev;
};
```

A new variable with the type of this struct could be given like:

struct linkedList exampleVariable;

The variable "exampleVariable" is laid out in memory like this:

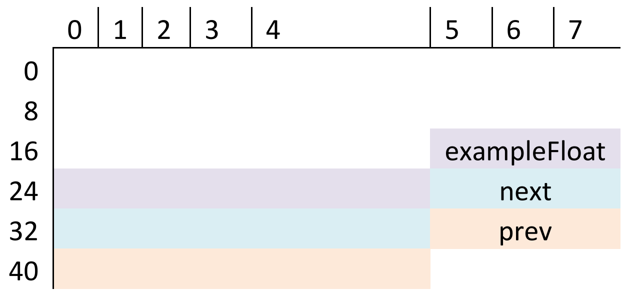


It will take up 45 bytes assuming the aforementioned sizes.

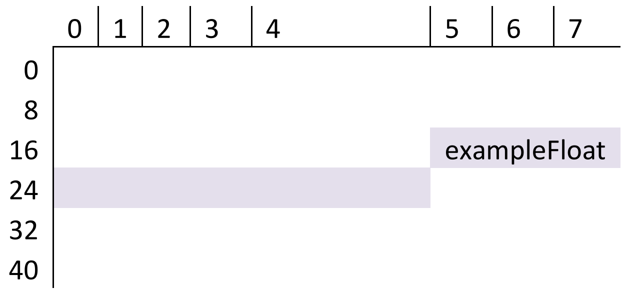
Though the struct may have no guaranteed default values, because it is a set of bytes rather than a pointer to a set of bytes, structs can be directly assigned to each other. This may seem obvious, however it is important to be aware that this has space and behaviour implications, for example, if you send the exampleVariable in to a function, let's say for the sake of example, a deletion function, and that deletion function sets the next node pointer of the structure we send in to the next pointer of the next node, and then frees the next node (the one that we took the next pointer from), we would see that the next node would be freed, however because we sent the structure into the function instead of a pointer to it, the structure would be copied into the stack space for that function and that copy would be modified rather than the original. As mentioned, this also involves copying the variable into the stack space for the function, which can take quite some time in some cases.

The . Operator

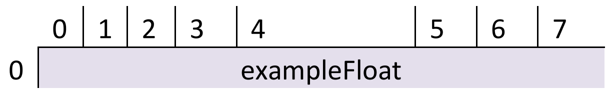
The struct would not be much use to us if we couldn't access any of it, which is where C helpfully comes to the rescue! C provides the . operator in order to dereference particular parts of our struct and gives us the values back. For example, using our struct from earlier, if we ask for exampleVariable.exampleFloat, C will essentially look up in a table where the "exampleFloat" part of the struct linkedList starts with respect to the 0th byte of the struct and then grab a number of bytes equal to the size of the type of the part is, so in this case, it will look up the "exampleFloat" part and see that it starts at the 21st byte, giving us this part:



It will then look at what type the requested part is, in this case double, and then see how many bytes that type is, in this case, 8 bytes, then grab that many bytes, giving us this part:



It then interprets the contents as a double and gives it to us:



We've now successfully retrieved that bit of the struct! We can also assign in a similar way,
exampleVariable.exampleFloat = 2;
Will set those bytes to the double value 2.0.

The -> Operator

As mentioned earlier, structs are entirely copied into a function's part of the stack and, as it happens, this is not often what people want, for various reasons. Because structs are so frequently passed by address, a new operator was made to replace the pointer dereference followed by struct dereference. For example, if we had another variable structPointer declared as:
*struct linkedList *structPointer = &exampleVariable;*
we could set the exampleVariable.exampleFloat value to 3 by first dereferencing the struct pointer to get the struct we what to edit (using *structPointer) and then getting the "exampleFloat" part of that dereferenced struct (using .exampleFloat), which when assembled would look like:
*(*structPointer).exampleFloat = 3.0;*
However, as mentioned, this operation happens so often that the makers of C deemed it appropriate to dedicate an operator to handling it, this operator is the -> operator. The operator replaces the dereference of the pointer followed by the struct dereference, so we could have instead set the value of exampleVariable.exampleFloat by using
structPointer->exampleFloat = 3.0;
Ah, how pretty.

Enums

The enum data type is a simple data type that has a number of possible, named, values. The underlying type is an unsigned integer, so takes 4 bytes. It is rarely used by most people and people who understand structs rarely have trouble using it, so no deep discussion will be assigned to it.

An enum is defined in a similar way to a struct, for example:
enum testEnum {TEST, TEST2, TEST3};
defines the enum testEnum type with the possible values TEST, TEST2, TEST3. A variable can also be created in a similar way to struct variables, for example:
enum testEnum exampleEnum = TEST;

Unions

The union data type is even more rarely used than enums, it is used in order to gain access to the same piece of memory a number of different ways, no conversion takes place and writes of one type will overwrite the other. A union is created and accessed in the same way as a struct. I think the primary case where you'd want to use a union would be cases where you needed a temporary storage space that could work for a number of cases that occurred independently. Regardless, they don't seem to be used very often.

Typedefs

A typedef is a macro that allows a new type to be defined from an existing one. Typedefs are often used to make code difficult to understand by hiding what types certain variables actually are, so be wary of overusing them, however they do appear and may be used in highly adaptable code to save many lines of changes. Typedefs are used by essentially writing a variable declaration and then including a #typedef in front of the declaration. For a very simple example, typedefs may be used to define data types like so:
*#typedef int *example_t;*
The type can then be used to create a new variable such as:
example_t newDataVariable;
which will be of type int *.

Memory Allocation Suite

It is worth noting that this can be looked up using man pages on UNIX systems, and a more in depth explanation is likely to appear there, explaining what flags are set on errors and such; however the points here are likely to correct misunderstandings about the malloc suite and explain the functionality. All of the allocation functions will return a NULL pointer if they fail. It is worth noting that code that performs differently on different machines is likely to have some kind of mistake to do with memory allocation and freeing (for the kind of programs expected to appear in this subject).

Sizeof

It is worth noting the sizeof macro, which is used extensively in allocation. The sizeof macro gives the size in bytes of a particular data type. A commonly made error is that early

programmers think that the sizeof macro does more than it does, if a malloc'd bit of memory is assigned to a pointer, using sizeof will return the size of the pointer (8 bytes on 64-bit machines), not the size of the allocated memory. Sizeof can be used on either types (such as sizeof(int)) or variables (such as sizeof(exampleVariable)), it should be used to determine the size of a data type, as that data type may vary between machines (for example the int type could be more or less bytes than the 4 seen on the machines we currently work on). If allocating space for an array of values, the usual form is to do something along the lines of:
sizeof(int)*numberOfElements
where numberOfElements is the number of values in the array, as this makes it clear.

Malloc

Malloc allocates memory on the heap and returns a pointer to it. Malloc takes a single argument, the number of bytes to allocate and hopefully returns the pointer to the requested space, if malloc can't allocate all of the space, it will set the ERRNO value to ENOMEM and return a NULL pointer, so the return value must be checked. It makes no effort to clear the memory that it allocates, so any junk could sit in the space allocated. It is the preferable method of allocation for a number of reasons, unless need your entire allocated space to be zeroed before you begin, you are better off using malloc than calloc.

Calloc

Calloc has very similar behaviour to malloc, however the arguments are different and the memory at the pointer returned (if not NULL) will be fully zeroed. The difference in arguments is that calloc instead takes two, the first being the number of elements to allocate and the second being the size of each element (though these are multiplied, so it doesn't really matter which is which).

Realloc

Realloc is the most versatile of the three allocation functions, and early programmers often make mistakes with assumptions involving it. Realloc takes two arguments, the first the pointer to the currently allocated memory for the variable, the second the size (in bytes) of the new region. Realloc may move the memory if it can't expand to the requested size where it is. If the given pointer is NULL, realloc will behave the same as malloc. Realloc will return the pointer to the new space if successful, because realloc can move the currently allocated memory; this is often changed, so ensure the new address is allocated. Realloc will return NULL if it fails to allocate the required memory; however, if it fails to allocate the memory, the original memory will remain in place and not be freed. Just like malloc, realloc will not do anything to the additional memory allocated.

Free

Just as we want to allocate memory, we also want to get that memory back and use it somewhere else after we're done with it. In order to help with that, we have the free function, the free function allows memory allocated on the heap to be reclaimed and used elsewhere. We simply need to send a pointer as a parameter to the free function and free will reclaim that memory. Important to note is that free cannot reclaim memory from the stack, so should not be used for that purpose, also important to note is that freeing memory that has already been freed has undefined behaviour, which could potentially range from the literally harmless to completely breaking the entire program. It is good practice to set freed pointers to NULL as well as pointers which are not allocated space without fail before being used.

Arrays

Arrays are specific data types in C, and are handled differently because of it, an array of values such as
int a[6];
creates space for a number of items (6 ints in this case), and allows them to be dereferenced by typing the right element number, for this example, a[3] would give us the 4th int (numbering starts from 0), and in fact, we could also get this value using *(a + 3). This behaviour is the same for pointers. However, arrays in C are not just shorthand for dereferencing. Considering two arrays
int aTwoD[3][3];
*int **pTwoD;*
where the second has been allocated using a loop to first allocate space for three int * values and then had space for three int values allocated for each of those int * values, the difference can become apparent, aTwoD[2][1] and pTwoD[2][1] can both give us the same value, as we would expect, however, the values we would get if we instead did aTwoD[0][3] and pTwoD[0][3] are not likely to match, aTwoD[0][3] is going to be the value aTwoD[1][0], while pTwoD[0][3] is likely to be some random value. The reason is because aTwoD is a flat 9 element array accessed via some mathematics involving the type of the array, while pTwoD genuinely follows the 1st element and then looks at the 4th element in that "row". This does mean that the two are ultimately incompatible overall, however being aware of this difference is important if it ever comes up. It is worth noting that because of these differences, as the array type in C is able to do away with pointers to spaces, the array type is more efficient at storing the same amount of data, so if you know you will be storing a certain amount of data, the array is the better choice. However, it is also worth noting that most of the time you don't know how much data you will need to store, so the dynamic array is often able to be smaller despite its inefficiency.

Right-Left Rule

The right-left rule is one that can be used to write and understand much more complex types without fear of a mistake. The rule is an algorithm that runs in four steps, those steps are:
1. Find the Identifier and then write or say
"<identifier name> is "
where <identifier name> is the identifier
2. Move right until reaching the end or a right bracket.
3. Move left until reaching the start or a left bracket.
4. Repeat steps 2 and 3 until complete.
While completing each step, * should be read as "a pointer to", [] should be read as "an array of" and () should be read as "a function returning".

For example, for the rather complex definition:
*int *(**(*p())[]);*
we would find "p is a function returning a pointer to an array of pointers to pointers to pointers to ints." Is what p is.

The rule is useful if you are unsure whether what you have written is exactly what you want the identifier to be.

Data Structure Examples

What is an Array?

An array is just a simple set of values. In C, arrays have a length known at run-time. For example, the set [1,2,2,3,4,5] is an array, [[1,2],[4,5],[7,8]] is also an array, speaking more generally than C, [A,[1,2],&func1,7,#35424] and similar are also arrays, as they are collections of items, in some sense in a row, it could be argued that they are also of the same type, however what constitutes a "type" is much more vague in that case.

Dynamic Arrays

A dynamic array is a data structure that is a type of array; generally a dynamic array has some kind of notion of how many items are in the array and how much space remains to add new items to the array. In C this is achieved by using a variable or two to keep track of these two properties, if there is a tight bound on the size (that is, the array is only enlarged when a new item is added and is shrunk when an item is removed), the two properties can be considered the same and a single variable to keep track of the size is all that is needed, however a much less computationally intensive technique is to have the two variables, one keeping track of the number of items in the array and one keeping track of how much space there is in the array and then doubling the array size after each time the array becomes full.

Worth noting is that dynamic arrays have benefits and shortfalls compared to hard-coded arrays, the benefit is obvious, that the array can be built so it will expand to fit the needs which are rarely known at run-time and avoid over-expansion. The downside is that dynamic arrays in C are implemented using pointers, so the pointer needs to be stored, also that the two variables indicating the state of the array must be stored, which also takes space, and, more significantly, the cost of expanding the array is not free and at times not cheap. Ultimately, if you know what the exact size of the array will be, going with the hard-coded choice is much better, however the rarity of that case means that the dynamic array is often the better choice by necessity.

Sorted Arrays

There are a number of ways to sort arrays, primarily being the obvious ascending or descending, however, there does exist other 'partial sorting' techniques that we can use. Sorting arrays allows us to significantly reduce lookup times, one of those lookup reduction techniques is the binary search, which essentially cuts the search space in half each failed comparison, finding a match, if one exists, in worst case O($\log_2 n$) time. It does this by looking at whether the desired key is above or below the centre of the array, and then splits the half the key could be present in in half again, comparing at the middle of that half, repeating until the key is either found, or the key cannot possibly exist in the sorted array.

Sorted arrays gain significant benefit at lookup time.

Unsorted Arrays

Though sorted arrays appear the best choice, what they're not so good at is insertion or creation, at least compared to their competitor. The sorted array inserts in O($\log_2 n$) time and the unsorted array sorting problem is inherently O($n\log_2 n$) (that is, the creation). However, an insert in an unsorted array is usually O(1) and the creation is usually also O(1), and lookup for a value is O(n), which is a lot slower, however, in the case that many insertions into a particular array are common and lookups are rare, the unsorted array may perform better than the sorted one.

Linked Lists

Though we touched on linked lists earlier, it may be worth covering them more formally, linked lists are like a set of chain links, links can easily be taken out from the middle and moved to the end. Primarily this makes deletion of particular "nodes" easy.
The linked list data structure is one which for every "node" (like a single link in the chain) includes at least a "next" pointer and some data for that node, often there will be a parent-like structure explaining where the "root" (the first link in the chain) of the list is and sometimes where the "tail" (the final link in the chain) is, sometimes the node will also include a "previous" pointer, which will point to the previous node. Lookup is done by starting at the head node and comparing the key data across each node until the tail is reached (the tail usually has a NULL pointer as it its next node, likewise, in the case that previous node pointers are present, the head node usually has a NULL pointer for its previous node as well), if the required key is not found, it is not in the list. Insertion is done at either the head or the tail, depending mostly on preference and available pointers. Deletion is done using a combination of lookup and setting the next node of the previous node to the next node of the node being deleted and the previous node of the next node being set to the previous node of the node being deleted (this sounds complex, but essentially this is just bypassing the node being deleted), then the node is freed.

Sorted Linked Lists

Linked lists do carry benefit from being sorted, in that at a certain point along the chain, a new value that is not present can be shortcut to being known to be not present earlier along the chain (usually). In either case, sorted or not, the complexity of a lookup remains O(n) for the linked list. Unfortunately the cost of insertion into a sorted linked list could be as bad as O(n^2).

The very expensive insertion cost may make you wonder whether it's worth even thinking about a sorted linked list, however, there's good reason to pursue the line of thinking because…

Binary Trees

Linked lists have a single link going to the next node and all linked lists look like they're particularly the best for is deletion, and the fact that we can't jump around the linked list checking values means that they don't gain as much as we would hope when sorted. However, if we add a second link to the next node and treat our list as a binary search on its own, we can get a lot of value out of the "linked list", this kind of multiple node choice "linked list" is called a tree and when we have 2 nodes specifically, it's known as a "binary tree". Using our binary tree, we can do something interesting where we place all nodes with keys greater than or equal to a particular node's key on its right and all nodes with keys less than a particular node's key on its left. This gives us a "rough" sorting that if we're lucky will give us a nice $\Theta(\log_2 n)$ lookup time.

Unfortunately, there are a few drawbacks with the vanilla binary tree, the first being that if all the values fall to one side we end up with a linked list again, however, this linked list ends up having an extra unused pointer, meaning we're wasting space. Such a bad outcome is probably rare for random data; however sorted data, or mostly sorted data, will produce this outcome. Another strange possibility is that all the values are the same, which could lead to very expensive lookups. The second issue is that keeping this tree balanced is not usually a highly cheap ordeal; however that's something we'll take a look at a bit later as there *are* nice ways.