COMP10002 - Foundations of Algorithms

Semester 1, 2017

Introduction to C Programming

> *C++ said to C: "You have no class!"*

C, is a *compiled language* and not an interpreted language (e.g. *Python*). That means, all C programs in plaintext need to be *complied* (converted to machine language) before they can be run. In addition to this, unlike Python, you don't need an external program to run a program that is written in C. You need to install Python to run a `.py` file. C files are converted to binary (`.exe` on Windows, and simply no extensions on UNIX-based systems like Mac OSX or Linux) files that can be executed directly.

Here's a simple C program to start with:

```c
/* This is a comment, the machine does not read anything out of it but it is very useful
to help other humans understand your code. */

#include <stdio.h>
int main(int argc, char* argv[]) {
    printf("Hello, World!\n");
    return 0;
}
```

This program, simply prints `Hello, World!` to the console and then exits immediately after that.

- `#include <stdio.h>` imports the **standard I/O library.** There are many other libraries that can be imported, libraries are provided as `.h` files (*aka 'header' files*).
- The `main()` function is the **entry point** of the program, all compiled programs in C will run the `main()` function first and all theother fucntions are referenced from `main()`.
- `printf("Hello, World!\n");` prints `Hello, World!` to the output stream. `\n` indicates a line break.
- All statements must be ended with a `;` (semicolon).
- `argv[]` is an array of the **program arguments** passed during the execution of the program.

Variables

Variables are used to store and manipulate values. In `C`, variables are declared using the syntax `<type> <name> [= value];`, for example:

```c
int answer;
float percentage =  51.2;
double pi = 3.14159265;
char crush_initials  = '?';
```

It is possible to declare variables without assigning any value to it, as shown in line 1 in the example above. The values can be assigned later when it is needed (through user input or otherwise). The variable types, as used above are:

- `int`: **int**egers ($\mathbb{Z}$)
- `float`: a **float**ing-point precision number, basically, a decimal.
- `double`: a **double**-precision floating-point number, like a `float` but has better precision in terms of calculations. Even so, there are cases where it is more efficient to use `float` and there are also cases where the use of double-precision numbers are strictly needed.
- `char`: single **char**acters. As a literal, they must be enclosed in **single** quotes. For instance, `'A'` represents the letter A.

There are some **object naming conventions** that must be followed when naming variables:

- Variable names can only contain *alphanumeric characters* (i.e. `ABC..YZ abc..yz 012..89`) and `_`. Other symbols cannot be used as part of a variable name.
- Variable names must **start** with an **alphabet**. While you can include numbers and `_` in variable names, you can only start variable names with the characters `ABC...XYZ abc..yz`.
- When referencing variables, they are **case sensitive**. So referencing `Var1` is not the same as referencing `var1` or `VAR1` or `vaR1`.

Input and Output

Input and output is provided by the `stdio.h` library. It provides the following important functions:

- `printf(char* format, object i1, object i2, ...)`: prints a formatted string with substitutions `i1`, `i2`, `i3`, ..., and so on.
- `scanf(char* format, object* i1, object* i2, ...)`: gets formatted input and store them to variables `i1`, `i2`, ..., and so on.

Example program with input and output:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    int answer1; double answer2;
    printf("Give me an integer followed by a decimal, please: \n")
    scanf("%d %lf", &answer1, &answer2);
    printf("You gave me: %d, %f \n", answer1, answer2);
    return 0;
}
```

This program simply echoes the user's input as a formatted output, if, say, the user inputs `7 2.0`, then the output will be:

```
Give me an integer followed by a decimal, please:
user> 7 2.0
You gave me: 7, 2.000000
```

In this program:

- `scanf` parses user input and stores them in variables `answer1`, `answer2` using **pointers**, (i.e. the `&` preceding the variable name). You will need to use to prefix all variable names with `&` when using

`scanf` or otherwise the program will encounter a runtime error.
- `printf` prints a formatted string by replacing the placeholders `%d`, `%f` with the variables `answer1`, `answer2`.

> **What are pointers?**
>
> While pointers are not yet discussed at this point, it is best for you to have a slight idea of what it is. Variables with and without pointers convey two different meanings:
>
> - `answer1` evaluates to the **value** of the variable `answer1`.
> - `&answer1` refers to the **variable slot/memory slot** `answer 1`.
>
> Hence the reason why we need to use pointers in `scanf` is because we want to tell the compiler to store the user input **in the variable slot** `answer1`. We're not trying pass the current **value** as an argument to scanf.

Formatting and String Substitutions

The `f` in `printf` and `scanf` stands for *format*. These two functions do more than just output text or input values to a variable but they do it in a specified format that you provide. In the previous example, printed substrings `%d` and `%f` are replaced with representations of `answer1` and `answer2` as an integer and a floating point up to 6 decimal places respectively, this is called **string substitution**. The substitution placeholders in C are:

- `%d`: represents the value as an integer. `d` here actually stands for "**d**ecimal", emphasizing that decimal numbers by definition are **numbers in base 10**, *not* numbers with a fractional part.
- `%3d`: print exactly `3` digits of the integers, you can replace the `3` with any value `n`.
- `%f`: represents the value as a `float`. By default, they are displayed with 6 decimal places.
- `%2f`: displays `float` or `double` values with 2 decimal places. You can replace `2` with any `n` to display `n` decimal places. The precision of `float` values are up to 20 decimal places.
- `%lf`: placeholder for `long`-type variables, not all systems support this.
- `%c`: placeholder for `char`-type variables.
- `%s`: represents the value as a `string`. (i.e. an *array of characters* or *text*).

Placeholders in the format are replaced in the same order as the order of the subsequent arguments. So for example:

```c
int a = -24;
int b = 16;
printf("My numbers are %d and %d.\n", a, b);
printf("My numbers are %d and %d.\n", b, a);
```

Noting that the arguments passed to the `printf` functions in line 3 and line 4 have reversed orders, the code above will print out the following output:

```
My numbers are -24 and 16.
My numbers are 16 and -24.
```

While this example does not change the English meaning of both outputs, be careful when using `printf` or `scanf` as different orders can result in different or incorrect interpretations of the output.

Suppose that you specified the `n` characters to print, but the actual value printed takes less than that amount of character space. Then, the console will print out leading whitespace such that exactly `n` characters are used to represent its value. For instance:

```
printf("%3c", 'A');
printf("%7d", 321);
```

This will print out the values with the leading whitespace, such that that line has exactly 3 and 7 characters. That means, line 1 will have 2 leading whitespaces and line 2 will have 4 leading whitespaces as shown below:

```
  A
    321
```

Constants

Constants are values defined in the beginning of the program, it used to store **values that will not be changed later in execution**. It can be used, and evaluated, as a variable with the exception that **you cannot assign or re-assign value** to constants. The syntax for defining a constant is as `#define name value`. For example:

```
#define PI 3.14159265
#define LANGUAGE 'C'
#define LECTURER "Jianzhong Qi"

int main(){
  double side;
  double area = PI*side*side;
  double circ = 2*PI*side;
}
```

Note that **we do not use semicolons** in definitions as similarly as we don't use semicolon when using `#include`, and he standard naming convention for a constant is to be *entirely capitalized*.

Unlike variables, constants do not use memory space as all statements that use constants are **evaluated during compilation**.

Manipulating Data and Variables

Integer and Floating Point Arithmetic

In C, you can use arithmetic operations to manipulate variables and other values.

The five binary arithmetic operations in C are: `+`, `-`, `*`, `/`, `%` - which they are used for addition, subtraction, multiplication, division and modulo (*remainder*) respectively. They can be used for all `float`, `double` and `int` data types. The results of an operation can be evaluated as a literal and does not need to be assigned to a variable, which means they can be used like this:

```
printf("%d", 19 * 2 - 3)
```

This will instantly print out the result of the calculation which in this case, is `35`. Arithmetic operations can be directly used on two variables, two literals, or a literal and a variable.

When there are no overloading definitions, operations exhibit a property that is mathematically known as **closure**. The result of an arithmetic operation will be of the same type as its arguments. In other words, operations on two `float` type will result in a `float` type and operations on two `int` type will result in an `int`-type (including divisions).

Increments:

Increments are a way to change the value of a variable by *exactly* `1`. The syntax is as follows:

- `var++` *increases* the value of `var` by 1.
- `var--` *decreases* the value of `var` by 1.

Computational Limitations

Integer Division

An `int` divided by an `int` will result in an `int`. Hence a division of two integers will result in a whole number even if it is not mathematically meant to be that way - however, the results will always be **floored** (rounded down) even if they are closer to the next number (so even 3.9 is rounded down to 3). In mathematical terms, if a and b are `int` types then `a / b` is equal to $\lfloor \frac{a}{b} \rfloor$ For example:

```
int a = 9; int b = 2;
printf("%d / %d = %d \n", a, b, a/b)
```

The above code will print out `9 / 2 = 4`, which is not mathematically true. This does not happen when you are dealing with `float` or `double` values since they natively support fractional parts.

Integer Overflow

The event such that an arithmetic operation causes an error by *wrapping around the number line*. This happens because the `int` type is limited in terms of length. Let `INT_MAX` be the highest possible number and `INT_MIN` is the smallest possible number (largest negative), hence `INT_MAX + 1 = INT_MIN`. For example, this is demonstrated below in C:

```
int big, bp1, bt2, bp1t2;

/* Lecturer's note: This is NOT how you would want to name your variables. You want your
variables to have a meaningful name so that other people that is reading your code - or
your tutor, would not be confused */

big = 2147483647;
bp1 = big + 1; // Big Plus 1
bt2 = big * 2; // Big Times 2
bp1t2 = bp1 * 2; //Big Plus 1 Times 2
```

Then the values of the variables are as follows:

```
big == 2147483647
bp1 == -2147483648
bt2 == -2
bp1t2 == 0
```

In C, `big` is the largest possible integer that you can have and hence `big + 1` will result in the lowest possible integer. Which is `bp1` in C.

Note that integer overflow is **silent**, meaning that **this error will not be reported by the compiler** even when enabling all compilation warnings.

The maximum signed integer in C is equal to 2^{31}-1, while the lowest possible signed integer is -2^{31}. The term *signed integer* here means the integers that can be positive or negative, in contrary, *unsigned* integers start from zero to up to 4294967295 (`0xffffffff`).

The values of the upper bound limits for both signed and unsigned `int`, and all other variable types are stored as constants with the name in the header file `limits.h`. The constants for signed integers are named `INT_MAX` and `INT_MIN`.

Limited Precision

`double` and `float` type variables have limited precision. They are really bad at representing number to exactly 100% precision. The main reason for this is because the computer internally stores all values in binary (in base 2) and base 2 is does not always work well at representing any number. In fact, base 10 also fails at making good representation of some numbers.

To see why computers are not good at calculating with real numbers, we can use the following example:

> In base-10 we have $\frac{1}{3} = 0.3333333\cdots$, even if computers store values in base 10 - they only have a **finite amount of memory and cannot store an infinite decimal expansion**, hence the computer is going to truncate its value to something like $\frac{1}{3} = 0.3333333$. So when we multiply by 3, we have $3 \times \frac{1}{3} = 0.9999999$ which is not true due to the precision error.

In terms or precision, the `double` type is still better than `float` as `double` uses more memory and are able to store way more decimal digits than `float`.

Example:

```
double x, y, z;
x = 0.1;
y = x + x + x + x + x + x + x + x + x + x;
z = y - 1.0;
```

Then the values of the variables would be:

```
x == 0.1000000000000000555112,
y == 1.0000000000000000000000,
z == 0.0000000000000000000000,
```

As you can see, the value of `x` is not exactly `0.1` as we assign them previously. However, they manage to precisely calculate `y` and `z` based on the value that *we* expect. However, with *the above value of* `x`, the value of `x + x + x + x + x + x + x + x + x + x` should be not exactly `1.0`.

Type Aliases (`typedef`)

`typedef` allows you to **create an alias** for a specific type. The syntax is simply:

```
typedef type alias;
```

It is hard to see the use to relabel `int` as something else, but they are **useful to rename collections**. For example, it is useful to call an array of 3 `double` type variables as `vector`. When so is done, you can define functions to output or take vectors as input. Of course, it is still impossible to output arrays per se, it still needs to be output as pointers.

```
typedef double[3] vector_type;
double magnitude(vector_type vec) {
  return sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);
}
```

Typedefs are much more useful when we learn about `struct` later.

Control and Program Flow

Conditionals (`if` statements)

Unlike in other languages like C# or Python, C does not have a `boolean` type and hence `true` and `false` are not pre-defined constants. Logical expressions will evaluate to the type of `int`. When using `if` statements, the conditional is read as:

- `0` is interpreted as `false`.
- Any non zero integer value is treated as `true`.

The syntax for `if` statements in C is:

```
if (condition) {
    /* Anything here will be executed if the condition is non zero */
} else if (condition2) {
    /* Anything here will be executed if condition is zero, but condition2 is nonzero */
} else {
    /* If all else fails */
}
```

Logical statements in general are evaluated to `1` if they are true and `0` if they are false.

Comparisons and Compounds

The syntax used to compare to quantities in C are:

Test	**Comparsion Operator / Syntax**
Equality ($a = b$)	`a == b`
Inequalities ($a > b, \quad a < b, \quad a \geq b, \quad a \leq b$)	`a > b`, `a < b`, `a >= b`, `a <= b`
Not Equal To ($a \neq b$)	`a != b`

> **Common Misconception**:
>
> Since logical statements evaluate into `0` and `1` and they are `int` types, logical statements can be directly (although unnaturally) compared with another `int` value. For instance, take a look at the following code:
>
> ```
> if (4 > 3 > 2) {
>   /* Do this only if 3 is between 4 and 2 */
> }
> ```
>
> It is easy to be tricked into thinking that this code actually checks whether $4 > 3 > 2$ which should evaluate to true (or to `1`), that is wrong. The conditional actually fails to be true here, and the code inside the `if` block will not be executed. As the boolean expression inside the if conditional are evaluated, they are evaluated from left to right. So at first, the computer will evaluate whether `4 > 3` is true and since it is in fact true, `4 > 3` will evaluate to `1`. Hence, now it continues to evaluate to the right and **the computer will try to evaluate whether** `1 > 2` **is true**, which is in false and will evaluate to `0` instead. In other words, evaluating `4 > 3 > 2` is equivalent to evaluating `1 > 2`. Chaining inequalities as an interval simply doesn't work because **this is not Python**.

The compounds used in the C language, according to their *order of precedence* are:

English	**Boolean Operator**
NOT (Negation)	`!condition`
AND (Conjunction)	`condition1 && condition2`
OR (Disjunction)	`condition1 || condition2`

Since `int` values are used here instead of native `booleans`, you can use the boolean operators on literal integers. For instance, `!n` will easily evaluate to `0` if `n` is non-zero and to `1` if `n` is exactly `0`. You need to use brackets when working with a conditional operator and negating it, for instance, `!(a>b)` to denote whether `a` is **not** greater than `b`.

Input Processing

The `scanf()` function returns an `int`, this can be used to process and validate input. The first argument of `scanf()` is a pattern that the user needs to match when entering the input, and the `scanf()` function returns the number of patterns matched by the user. So to validate inputs, one can use the following set of code:

```
if (scanf("%d %d", &v1, &v2) == 2) {
    /* code if input is valid */
} else {
    /* code if input is NOT valid */
}
```

As you can see the above code asks the user to input **exactly two integers**. So if the user types two integers separated by a whitespace (it can be a space or a new line), both `%d`s in the pattern will be matched there will be 2 matches, so `scanf` will return 2 and the code block for the valid input will be executed.

Note that further validation of input may need to be done, this can be done by using *nested if statements*

Selection (`switch` statements)

The selection is just a more sophisticated version of the `if` statement that allows many conditionals at once. The syntax for a `switch` statement is as follows:

```
switch(variable) {
  case a:
    // code execution will START here if variable equals a.
  case b:
    // code execution will START here if variable equals b.
    break; //used to exit the entire switch statement.
  default:
    // code execution will START here if there is no match to other cases.
    break;
}
```

The value of `variable` will decide where does the exececution starts within the `switch` statement. Note that the program will continue to flow within the `switch` statement if `break` is not used. In the example above, when `variable` equals `a`, the code in `case b` will also be executed since there is no `break;` in `case a`.

Unlike `if` statements, the `switch` statements **can only be used for equalities**. You cannot use to replace other kinds of condition like an *interval* (`<`, `>`, `>=`, `<=`).

Iteration

`for` Loops

The `for` loop has two parts, a *head* and a *content*. The syntax for a `for` loop is as follows:

```
for (init; guard; post){
  /*
    This part is the 'content' of the loop. All statements here will be evaluated as
long as condition evaluates to nonzero.
  */
}
```

The head part (the part inside `()`) has 3 parts, and they are:

- `init` is executed before the first execution starts, it is usually used to initialize increment variables.
- `guard` is a conditional expression needed in order for the loop to start executing. The loop will stop **once the condition evaluates to 0**. Often it needs to be trivially true at the first iteration.
- `post` is executed after each execution of the loop content.

An example of for loop:

```
for (int i = 0; i < 100; i++) {
  printf("%d", i);
}
```

This code simply prints the numbers from `0` to `99`. First by initializing the value `int i = 0`, then printing `i` over and over again until the `i < 100` is not satisfied while incrementing the value of `i` for every time it finishes printing.

`while` and `do` Loops

`while` loops are simpler, the syntax are as follows:

```
while (condition) {
    /*
    This part is only executed when condition and for as long as it is true
    */
}
```

The structure makes it easier to be used when we want something to keep executing for as long as a condition is true. When the `condition` is false **once**, the loop terminates and the while loop will not be revisited again.

A similar loop to `while` is the `do` loop:

```
do {
    /* do stuff here */
} (condition)
```

The only difference between the `do` loop and the `while` loop is that the `do` loop will always **execute at least once** and then the block of code will be repated over and over again as long as the condition is true. `while` loops will not be executed when the condition is trivially false at the beginning of the `while` statement.

`exit()`

For many reasons that the program may need to be terminated, the best way to do this is using `exit(EXIT_FAILURE);` provided in `stdlib.h`. This function immediately terminates the program and tells the operating system that the program has exited in failure.

Functions and Abstraction

Functions are mainly used so you can repeat a process without copying the same code over and over again. Furthermore, functions provide a form of abstraction, such that you can **hide the details of execution**.

A function takes some values as input (called **arguments**) and then evaluates to an output, usually the output varies accordingly to the input to the function. A function is **defined** as:

```
<ftype> function_name(<type 1> arg1, <type 2> arg2, ...) {
    /*
        some code here that manipulates the inputs
    */
  return value;
}
```

Where `<ftype>` is the type of the output value, and `<type n>` is the type for the n-th argument. It is also possible to have a function that does return anything, such functions have the return type `void`. Such functions are usually executed for their side effects.

Now, just like variables, functions need to be declared (**not defined**) before they are used. For example, if your function is used in the `main()` function then the function must be **declared before** the `main` function. **Defining a function declares it as well**, so if a function is defined before `main()` a declaration will not be needed, but if it is defined *after* `main()`, a declaration will be needed before the `main()` function and the syntax for a declaration is as follows:

```
<type> function_name(); //This is the declaration.

int main() {
    function_name(arg1, arg2); //This is a function call.
    return 0
}
```

Note that **this does not mean all functions need to be declared before `main()`**, but simply before the function that they are used in. Often it is needed that a function executes another function in its definition. Calling a function will evaluate to the value that it returns and a function terminates as soon as it hits `return`.

Scopes

Variables defined within a function is only **local**, this means that you cannot access those variables from outside the function. A variable with the same name defined in **two different functions** do not share their values. Variables declared outside any function (even outside `main()`) is **global**. That means it can be accessed from any function and anywhere within the program. Function arguments are local variables.

Conceptually, two variables with the same name **does not share the same memory address**. Local variables are stored in a *stack*, where global variables are stored as *data* in the memory.

```
int var1 = 0;
void func_1() {
  int varA = 4;
  varB; //This will fail.
  var1;
  varA;
}
void func_1() {
  int varB = 5;
  varA; //This will fail.
  varB;
  var1;
}
int var2 = varA + varB
```

Static variables are initialized in a function, **but they do not get destroyed when the function finishes executing**. The value of **static variables** only persist when the same function is called, they are not accessible globally. One notable use of a static variable is to *keep track of how many times a function has been executed*.

All **local** variables are destroyed whenever the function finishes its execution. **Global and static variables should be avoided** in general to prevent side-effects and unwanted program behavior, but they are useful for making program-wide configurations.

Recursion

A recursive function is a function that **calls itself** as part of its execution. SA simple example of a function that can be written recursively is the function that is given an integer `n`, calculates the sum of all the natural numbers up to `n`.

```
int sum_natural(int n){
  if (n <= 0) {
    return 0;
  } else {
    return n + sum_natural(n-1); //The function calls itself here
  }
}
```

The above works since if we denote the sum from 1 to n as S_n, when we have the recursive relation $S_n = n + S_{n-1}$. Given the base case $S_1 = 1$.

Note that the argument has changed from `n` to `n-1`, it is important to call the function on a **different argument** than the original one or else the program may end up being stuck in an infinite recursion.

When calling a recursive function, each step of the recursion is done on a separate *stack frame*. Variables are not shared between two functions in a different stack frame.

Pointers

Pointers are used to directly access and manipulate memory. The simplest pointer operations are the **referencing** (`&`) and **dereferencing** (`*`) operators. Assume `var` is a variable, then:

- **`&var`**: Accesses the **memory address** of the variable `var`. Does not evaluate to its value. When printing memory addmresses, it is best to print it as a hexadecimal number.
- **`*address`**: `*` is used to declare a variable that will be used to **store memory addresses** on declaration.
  When evaluating instead of declaring, `*address` evaluates to the **value stored in the address** while `address` simply evaluates to the memory address.

- `+`: Used to **add memory addresses** and integers, so for example if we want to access the next address after `ma` we would use `ma + 1`.

Pointer variables have their types derived from regular variables, `int*` means "**pointer to `int`-type variable**". Untyped pointers are allowed, and they have the type `void*`.

Pointeres and direct memory access are used when we want to make changes made inside a function persist permanently.

Pointer Algebra

Let `p` be of type `type*` and `k` is an integer, then:

- The syntax `p+k` returns the address `p+k*sizeof(type)`.
- The syntax `p[k]` is equivalent to `*(p+k)`. It dereferences the address `p+k`.
- The syntax `p++` is equivalent to the syntax `p+1`, so it returns the address `p+sizeof(type)`.
- The syntax `p->property` takes the pointer of a specific property of a struct, then dereferences it. *explained further later*.

Arrays

Array is a **finite** collection of elements of the **same type**. In fact, arrays do not actually **store their values** inside itself, but it stores it in other memory addresses and the arrays **act as pointers to those addresses**.

```
int A[N]; //initializes an array A of N elements.
A[i] = n; //change the i-th element of the array to n. <*>
A[k]; // Evaluate the k-th element of A. <*>
```

> $\langle * \rangle$ Note that $i, k \in [0, N)$, since computers start counting from zero.

As **arrays only act as pointers**, `A` is in fact a pointer. Hence evaluating `A[i]` or getting the `i`-th element of a is equivalent to getting the value at memory address `A + i`. In other words `A[i]` is equivalent to `*(A+i)`. However, **unlike pointer variables**, they are **not assignable** (i.e. they are **constant pointers**).

It is also possible to assign array to pointer valued variables, it is **valid and in fact safe** to do the assignment `int *pointer = A` when `A` is an `int[]` type.

Multi-Dimensional Arrays

May be interpreted as an **array of arrays**, a table or a matrix. However, it can't be interpreted entirely as a table as **arrays are just pointers**, so array of arrays is a more appropriate approach. The declaration syntax is simple:

```
int A[ROWS][COLUMNS]; //initializes an array of ROWS arrays, which contains COLUMNS
integers.
A[i][j]; //evaluates to the j-th integer of the i-th array in A
A[i] //evaluates to the i-th array in A, returns an array/an array pointer
```

Of course the above only generates an array of arrays (i.e. it's limited to two dimensions). You can go as deep as you please in the dimensions by using `int[][]..[]` to generate an array of arrays of arrays of $\cdots$ of arrays.

Passing Arrays to Functions

Arrays cannot be passed as an argument to function. Instead, arrays passed as arguments to functions will be interpreted as pointers to the `0`-th element.

The main difference here is that since **arrays are not arrays anymore** in function, it is impossible to obtain the size of an array argument using the `sizeof` function.

Strings

There is no pre-defined `string` type in C. Strings are **null-terminated array of characters**, strings are stored as `char[]` type. Due to its nature as an array, **string variables are also pointers**.

Common libraries for string manipulations: `ctype.h`, `string.h`.

Strings of size `n` needs to be stored in an array with size `n+1`, with the last character being the null-character (denoted and written literally as `\0`).

String limitations in C:

- No literal reassignment, you can't immediately say `str = "some string right here"` (when `str` is already initialized). This is because they are arrays/pointers.
  Instead, copy each of the characters one by one. Some functions are available in `string.h` to acommodate this.

Initializing strings:

```c
char str[1000] = "Hello World!", str[1000]; //The latter initializes an empty string.
```

Geting strings from the user:

```c
char str[1000];
scanf("%s", str); //we do not use '&' here since str is an array therefore a pointer
```

Note that we do not need to to use the reference operator `&` the variable because character arrays already **interpreted as** pointers.

String Manipulation

Many functions are provided in `string.h`, the most commonly used being:

- `strlen(char* string)`, calculates the length of the string (excluding the null byte).
- `strcmp(char* s1, char* s2)`, calculates the 'difference' between `s1` and `s2`:
  - If it returns `0`, the two strings are equal.
  - If it returns something **less than zero**, then `s1` should come **alphabetically before** `s2`.
  - If it returns something **greater than zero**, then `s1` should come **alphabetically after** `s2`.

  Alphabetically meaning according to the character's ASCII values.
- `strcpy(char* dest, char* src)`: copies the string in the `src` to `dest`.

Note that all these functions require a string to be **properly terminated** to function. As they look for `\0` to know when to stop.

Analysis of Algorithms

Correctness and Termination

To analyze the correctness of a program, we use **assertions** on specific points in the program. Assertions **specific conditions that must be always true in a point of the program**. We can use the `void assert(int condition)` (from the `assert.h` library) - the program will halt when the `condition` passed to `assert(condition)` is false.

Assertions are also used to **prove** the correctness of a program.

Psuedocode Example (Linear Search):

```
for (int i = 0; i < N; i++) {
  assert i < N; //At this point the counter must not be beyond array bounds.
  if (A[i] == query) {
    assert A[i] == query; //we can only return a match, hence this assertion.
    return A[i];
  }
}
assert i >= N; //we can only return not found IF we searched through everything, hence this assertion.
return "not found"
```

Termination

It is also important that **we make sure that our algorithm will terminate**, this can be done using assertions similar to how we analyze for correctness.

Using the same linear search example above:

- We see that we start with a loop with `i` from `0` to `N` whenever `i < N`.
- Now since `i` is incremented every iteration - we know that soon that `i` is going to exceed `N`.
- So the for loop **will terminate** and the algorithm will terminate as well (since there are no loops after that).

It is **really** important that our algorithm will not go through an infinite loop, given any input.

In **recursive functions**, it is needed that every input will eventually reach any of the base cases given.

Efficiency

Efficiency here refers to the amount of **time** needed for an algorithm to complete. Time here is done in terms of **number of comparisons/calculations/iterations/steps** to complete.

Depending on the input, there may be different efficiencies for the **best case** or **worst case** scenarios of the input. For example, the linear search is easy when the item we look for is the **first item**, but the worst case scenario **is when the item we look for is the last one**.

The important factor in efficiency is the **rate of growth** of the time taken as the input size varies, not the exact runtime of the algorithm.

Big-O Notation

We use the big-O notation to denote the rate of growth of a function. In algorithm analysis, we use it to see how the runtime of algorithm grows as the **input size n varies**. The definition of a class of function $\mathcal{O}(n)$ is as follows:

$$f(n) \in \mathcal{O}(g(n)) \iff \exists N, c > 0 : \forall n > N \quad f(n) \leq c \cdot g(n)$$

Essentially, $f(n) \in \mathcal{O}(g(n))$ only means that **the function $f(n)$ is bounded from above by a multiple of $g(n)$ for sufficiently large values of n.**

We read it aloud as "$f(n)$ is of order $g(n)$"

This also means **the growth is bounded**, so the runtime an algorithm that runs on $\mathcal{O}(g(n))$ *'grows like'* $g(n)$, when n is the input size. We always simplify the 'order function' (i.e. $g(n)$) as simple as possible. While it is right to say that $5n + 2 \in \mathcal{O}(2^n)$ since the growth of n is always bounded by a multiple 2^n, it's much better to say $5n + 2 \in \mathcal{O}(n)$ as it reflects the growth of the function $5n + 2$ more accurately.

Since we want our algorithms to run really fast even for large inputs, **we want our time function of input size to grow really slowly**.

Classifying Functions:

The most practical way to classify a function with the big-$\mathcal{O}$ notation is simply to take the **most dominant term** (the term that grows the fastest) of the function, and then set the coefficient to one.

Note that the class of common functions, sorted from the slowest growth to the highest growth are as follows (does not include every thing):

$\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(\sqrt[k]{n}) \subseteq \mathcal{O}(n^k)$[1] $\subseteq \mathcal{O}(a^n)$[2] $\subseteq \mathcal{O}(n!) \subseteq \mathcal{O}(n^n)$

- The class $\mathcal{O}(n^k)$ is called **polynomial time**, higher values of k are more dominant. The order above applies for $k \geq 1$.
- The class $\mathcal{O}(a^n)$ is called **exponential time**, higher values of a are more dominant. The order above applies for $a > 1$.

Search Algorithms

Linear Search

To look for an element `k` in an array `A`:

- Start `i = 0`.
- While `i < size(A)`:
  - If `A[i] == k`, you are done. Return the position of `k` (`i`).
  - Increment `i`
- If you reach this point, `k` is not in the array.

Implementation of linear search in C:

```
int search(object k, object A[], int size) {
  for(int i = 0; i < size; i++) {
    if (A[i] == k) return i;
  }
  //k cannot be found
  return NOT_FOUND;
}
```

The worst-case and average-case complexity of linear search is $\mathcal{O}(n)$. This is quite sustainable for small to large sized inputs, but not **very large** (in terms of array size) inputs.

Binary Search

To look for an element `k` in a **sorted** array `A`.

- Check the middle element of `A`, that is `A[n/2]`
- If `A[n/2]`, you're done. Otherwise:
  - If `k` < `A[n/2]`, perform binary search on the sub-array from `A[0]` to `A[n/2 - 1]`.
  - If `k` > `A[n/2]`, perform binary search on the sub-array from `A[n/2 + 1]` to `A[n-1]`.

The average case complexity of the binary search is $\mathcal{O}(\log n)$, much better than linear search.

Although we need to sort the array first, which can only be done at best at $\mathcal{O}(n)$ complexity, or most often at $\mathcal{O}(n \log n)$ complexity.

Hence to perform a binary search on an unsorted array it will take at average $\mathcal{O}(n \log n + \log n) = \mathcal{O}(n \log n)$ complexity. **Worse than performing linear search**. **This does not render the binary search redundant**, as for instance, you will be able to perform multiple searches faster by sorting an array once and doing binary search multiple times to look for different items to reap the benefits of the logarithmic complexity. On the other hand, linear search not really good at this.

Implementation of binary search in C:

```
int bsearch(object k, object A[], int first, int last) {
  int mid = (last-first)/2;
  if (A[mid] == k) {
    return k;
  } else {
    if (first == last) {
      return NOT_FOUND;
    }
    if (A[mid] > k) {
      return bsearch(k,A,mid+1,last);
    } else if (A[mid] < k) {
      return bsearch(k,A,first,mid-1);
    }
  }
}
```

Sorting Algorithms

Insertion Sort

Given an unsorted array `A`, the insertion sort works as follows:

- Initialize a new empty array `R`.
- While `A` still has elements:
    - Add the first element of `A` as the last element of `R`. Now `R` has size `k+1`.
    - If `R[k-1] > R[k]`, swap `R[k-1]` and `R[k]`. Otherwise `R` is now sorted.
    - Add the next first element of `A`, if such a thing exists.
- If `A` is empty, then `R` is `sorted(A)`.

While this algorithm works really well, the **worst-case and average-case complexity of insertion sort** is $\mathcal{O}(n^2)$. It is pretty good considering it runs on **polynomial time** - but would want something faster than quadratic time when working with really large data.

```
void sort(object[] A, int size) {
  object B[size];
  for (int i = 0; i < size; i++){
    B[i] = A[i];
    for (int j = i; j > 0; j--) {
      if (B[j] < B[j-1]) {
        swap(B+i, B+j);
      }}}}
```

Quicksort

Given an unsorted array `A`:

- If `A` only has 1 element, you are done and the array is sorted. Otherwise, continue.
- Choose a random element in `A`, call it `k` (this is the **pivot**).
    - Initialize two new arrays `Kp` and `Kl`.
    - Put all elements lower than `k` to `Kl`, and all elements greater than `k` to `Kp`.
    - Use `quicksort` recursively on `Kp` and `Kl`.
- Concatenate `Kl`, `{k}`, `Kp`.

The worst-case complexity of quicksort is still $\mathcal{O}(n^2)$, but its average-case is $\mathcal{O}(n \log n)$. The worst-case complexity is also really rarely encountered and this is often attributed to bad random selection of the pivot, to be exact only if every other element of the array are all larger than or all smaller than the pivot chosen for all recursive call.

Quicksort is available on `stdlib.h` as `qsort()`

Mergesort

A sorting algorithm that works by splitting an array, sorting them separately, and merging them back together to a single sorted array (our goal).

Given an unsorted array `A`:

- **If `A` has one or no elements then it is sorted**, so we are done.

- **Split** `A` to into two equal (*equal*: the length difference of the partitions is at most 1) partitions `P1` and `P2`.
    - Mergesort `P1` and `P2`.
- **Merging Part**:
    - Start `i=0`, on every iteration:
        - Set `A[i]` to $\min($ `P1[0]`, `P2[0]` $)$.
        - Pop the element taken from the partition.

The worst and average case complexities of merge-sort is $\mathcal{O}(n \log n)$, note that the worst-case is **better than quicksort**. However, mergesort uses memory space so it has a **space-complexity** of $\mathcal{O}(n)$.

Heapsort

To do **heapsort**, there is one special data structure to do these called **heaps**:

Heaps are essentially a version of a **binary tree** that is flattened to an array. That is, an array defined with the following properties:

- It is visualized as a **tree**, where the root is `A[0]`.
- The children of `A[i]` is `A[2*i+1]`, `A[2*i+2]`.
- No parent may be smaller than any of its children.

Now to do the sorting itself, one has to:

- Create the **heap** `H` from the given array. This heap has a **corresponding tree** `T`.
- Swap the first and the last element of `H`, do the same with the corresponding tree `T`.
- Pop that element from `T`, call this new tree `subT`. Let `subH` be `{H[0], H[1], H[2], ..., H[N-1]}`.
- Perform heapsort on `subH`, noting that an array is sorted if it has one element.

The average-case and worst-case **time-complexity** of heapsort $\mathcal{O}(n \log n)$:

- Creating a heap takes $\mathcal{O}(n)$.
- Popping the last element in the heap (binary deletion) takes $\mathcal{O}(\log n)$.
- Since we pop the element everytime we create the heaps, the total time complexity is $\mathcal{O}(n \log n)$

Pattern Search Algorithms

Problem: Given a text sequence `T` and a pattern `P`. Does pattern `P` appear as a continous subsequence of text `T`?

Naive Search

Traverse through the entire input and check if the current subsequence is a match. Using the `strcmp()` function from `string.h`, we can do the following.

Let `str`, `substr` be of `char*` type, where we want to see if `substr` occurs in `str`.

- Let `char* s = str`.
- While `s < sizeof(str)-sizeof(substr)`:
    - If `strcmp(s,substr) == 0`, return `true`.

    - Else: `s++`.

The complexity of the naive string search is $\mathcal{O}(mn)$.

KMP Search

Define a "failure" function `F`. Let `s`, `i` be indexes for the text string and pattern strings respectively - whereas `n` is the size of the superstring and `m` is the size of the substring.

- `s`, `i` is now 0.
- While `T[s] == P[i]`.
    - Increment `s`, `i`.
    - If there's a mismatch (`T[s] != P[i]`):
        - Now `s = s + i - F[i]`.
        - `i = F[i]`
        - Start searching from position `s`.
    - If `s == m`: string is found.

The complexity of KMP-Match is $\mathcal{O}(m+n)$.

Indexing

Used for really long sequence of `T`, such that $n << m$. Remember `findrepeats.c`. Indexing is used by large-scale search engines such as Google and Bing.

There are many ways to perform indexing, but only one is covered entirely on this subject.

Suffix Array

One way to index something is to create a **sorted suffix array** of `T`. Then perform binary search of `P` on `T`.

Example: Take the string `motherlode`.

Then we can have the suffix array:

```
char* suffix[] =
{"motherlode", "otherlode", "therlode", "erlode", ..., "ode", "de", "e", ""};
```

Now to search whether a string `P` is a substring of `"motherlode"`:

- Perform a binary search on the suffix array:
    - Instead of checking of `P` matches the current entry of the array being checked, check if `P` is a substring of the entry.

This is ideal to prepeare to perform multiple searches on really large data. However note that suffix array has a tradeoff: large memory usage.

The complexity of searching (without indexing) is $\mathcal{O}(\log n)$. And for each string comparison (to check whether a string is a match or not we needs at most m comparisons). Hence the final worst-case complexity is $\mathcal{O}(m \log n)$.

Dynamic Memory Allocation

Allows the programmer, YOU, to dynamically change and allocate how much memory is used for certain variables according to your needs.

C provides functions that allows you to dynamically allocate memories to a specific variable, and allows you to manually manage memory.

Allocation: `void* malloc(int size)`

doc: This function creates `size`-bytes of memory and **returns a void pointer** to the first allocated cell.

`malloc` will return `NULL` when the requested memory size is not available (i.e. the computer has less free memory then `size`-bytes).

Memories allocated with `malloc()` are stored in *heap* instead of *stack* segment. Allowing them (pointers to them) to be returned by a function as they are not destroyed with the local function scope.

Use-case example:

```
/*  This allocates memory with size equal to n*(size required per type).
    Just enough to store n-ints, n-chars respectively.
    */

int *array; int n;
array = (int*)malloc(sizeof(int) * n); //Note that these have to be type-casted.
array = (char*)malloc(sizeof(char) * n);
assert(array != NULL); //fail-safe in case we don't have enough memory.
```

Destruction: `void free(void* pointer)`

`free()` is used to de-allocate memory as allocated previously by `alloc()`.

Use this to minimize memory leak and free up some RAM.

Re-allocation: `void* realloc(int size)`

This function **reallocates memory** by changing the memory size allocated to a certain pointer.

Useful when we (or the computer) realizes that we do not have enough memory for our purpose and we need more, allowing the creation of arrays not of fixed size, but arrays that grow as we need it to grow. This is called *dynamic-sized arrays*.

Data Structures

Declaring Structures

Structs allow you to define your own types in many ways. An **array** may only allow you to store homogeneous values (same `type`) but a **struct** allows you to store variables with heterogeneous type.

A good example would be "how do we store a *menu*?". It contains the item name (`char[]`) and the price of the item (`int`), but arrays are not sufficient to do this. Hence, a `struct` is needed. Their use are *parallel* to how we use classes in better programming languages.

Use-case example:

```
typedef struct {
  char name[]; /* Name of the item */
  double price; /* Price of the item */
} item_t;

/* Assign the values in the same order as you declare them in the struct declaration */
item_t banana = {"Banana", 4.55};
item_t latte = {"Latte", 4.9};
```

This creates a **type** called `item_t` and creates two new items, `banana` and `latte` respectively.

Now, to **select a specific property of the struct** one can use the selection operator `<struct>.<property>`. That is:

```
int latte_price = latte.price; /* evaluates to 4.9 */
int total_price = banana.price +
  latte.price; /* evaluates to 9.45 */
```

Structs as Pointers

Like other variables, you can have struct pointers. This can be used to modify the contents of an existing struct (e.g. using `scanf`).

However, the selection `.` operator does not work in the pointer world, as it is used to obtain values. In order to access the memory address of a property of a struct, given the pointer to the struct, we need to use the `->` operator.

Example:

```
void inflation(item_t *item) {
  //Doubles the price of an item, given the item_t struct declared above.
  int price = item->price;
  price *= 2;
}
```

The `->` structure takes a pointer, a property name, and will **return the value pointed**. Most importantly, values accessed in this way can be modified.

Recursive Structs

Sometimes you will need a `struct` with an element that is of its own type. A good example would be a **city** as follows:

```c
typedef struct {
  char *name;
  city_t north_city;
  city_t south_city;
  city_t east_city;
  city_t west_city;
} city_t;
```

As you can see the this `struct` contains the data of a city with the cities adjacent to it - where they are all of `city_t` types. However this piece of code wouldn't work because the compiler does not know yet what is a `city_t` while initializing the struct.

However, the following implementation will work:

```c
typedef city city_t;
typedef city {
  char *name;
  city_t north_city;
  city_t south_city;
  city_t east_city;
  city_t west_city;
} city_t;
```

Computers need to have systematic ways to store data and we design them.

The structure will need to support the following operations:

- `structure_t createNew`
- `item_ptr search`
- `structure_t insert`
- `structure_t deleteItem`

Linked List

Linked lists are a way to store linear data, just like an array. However, arrays are stored in large chunks of memory while Linked Lists are scattered everywhere in the memory.

They are made up of connecting nodes of two values `value` and `next`, the `value` holds the value of that following node and the `next` variable points to the next element in the linked list.

Search operations is of $\mathcal{O}(n)$ average time complexity, while insert operations are $\mathcal{O}(1)$. These complexities are similar to an array. However, unlike an array, appending two linked lists (given the `list` structure is defined) take $\mathcal{O}(1)$ average time.

Structure of a Linked List:

- `node` structure:
  - `value` containing the node's value.
  - `next` containing a pointer to the next node.
- `list` structure:
  - `head` containing pointer to the first element in the list
  - `foot` containing the last element in the list.

Stacks and Queues

Stacks an queues are special implementations of linked lists or arrays that enforces its elements to be accessed and in a specific order, depending to the order that elements are stored.

Similarly to linked lists (as they only a special implementation of them). The average case-complexities of insertion and searching are also $\mathcal{O}(1)$ and $\mathcal{O}(n)$ respectively.

Stacks (Last-In = First-Out):

- Can only insert on `head`.
- Can only access and delete `head`.

Queues (First-In = First-Out):

- Can only insert on `tail`.
- Can only access and delete `head`.

Trees

Trees are similar to linked lists, but instead it every node as **multiple `next` nodes** (called children). Also, there is no set `foot` element since we have multiple branches that leads to different `foot`s. Instead we can only access the "`head`" element, but it is called `root` in this context. In binary context, both search and insert operations is of $\mathcal{O}(\log n)$ average time-complexity.

Terminologies:

- The first element is called `root`.
- Nodes may have `children`.
- Nodes with the same parents are called `siblings`.

Trees are useful in many cases, such as to perform binary search using binary search trees. Since the `nodes` may contains other nodes that may contain further nodes, we can treat as if everything else beyond that node as a single tree. Hence, we can 'say' that `trees` contain `trees`.

Implementation of a tree structure:

```
typedef struct {
  void *value;
  node_t *left;
  node_t *right;
} tree_t;
```

This implementation requires us to 'manually' assign the left and right children of a tree.

Another implementation that utilizes function pointers:

```
typedef struct {
  node_t root;
  int* (*cmp)(int *a, int *b);
} tree_t;
```

This implementation allows us to pass in a specific 'comparison function' to arrange the order of which the children are arranged. Allowing us to have trees in ascending, descending or any orders. This is abstraction at its finest, see also the ***Function Pointers*** *section.*

Elements are searched, accessed and placed into trees **recursively**. To search a specific element in a **tree** data structure we have to search (the correct) **sub-tree**, and its sub-trees and so on and so on.

The main benefits of using a tree structure:

- Data is always stored and accessed in a sorted manner. Every data is always at the correct relative position.
- Searching data on a binary tree is $\mathcal{O}(\log n)$ complexity using binary search.
- You can attach a tree to a tree.

Some disadvantages:

- The tree may not be balanced, searching for specific items may take way longer than other items. (although often still $\log n$ time complexity)
- If items are put into the trees in sorted order, then the tree **degenerates into a linked list**. (or even worse, since we don't have the `list` structure and we don't keep track of the `head` and `foot` elements.)
- Degenerate trees are problematic, since searches will now take $\mathcal{O}(n)$ time instead of $\mathcal{O}(\log n)$ time.

Graphs

Graphs are arguably one of the most complex data structures. Usually used to model networks and/connections.

Graphs are a collection of nodes with paths between them, but they do not necessarily have a first or last elements.

Nodes in a graph has have two elements:

- `value`, the value that the node holds.
- `paths`, a collection of the nodes that can be accessed directly from this node.

Dictionaries

We can design a **hash function** `int h(x)` that maps the data to a seemingly random integer $[0, N-1]$. Where N is the size of the array we prepare for this, and $N > n$ where n is the number of data we have.

Then to search or insert a specific item, we can index our array at `A[h(x)]`.

Since indexing takes $\mathcal{O}(1)$ time, both search and insert operations will take $\mathcal{O}(1)$ time worst case (therefore $\mathcal{O}(1)$ time average case).

We need to choose the proper hashing function `h(x)`. Need to be careful, **most options for the hashing function `h(x)` will have collisions**, that is where `h(x)` returns same values for two different values of `x`.

Consider the **hashing function** `int h(x) = x % 5` (*this is a very bad hashing function*), where we put our values into an array of then we can see an example of a simple collision.

- If we want to map the data `4`, then we map it to index `h(4) == 4`
- If we want to map the data `9`, then we map it to index `h(9) == 4`

Hashing Methods

First, we need to design a good hashing method. Common good hashing methods use a combination of arithmetic and bitwise functions.

Let `int h(type x)` be our hash function with **range** `k`, **a hashing function is expected to satisfy the following**:

- **Deterministic:** `h(x)` may look random but may not include any random calls. For the same value of `x`, `h(x)` may not output different values when called multiple times.
- **Uniform:** since `h(x)` has a range `k`, each value `0, 1, 2, ..., k` must have equal probability of appearing even with all possible different.
  This is 'not compulsory', but hashing functions that are not uniform can rather be problematic.
- **Surjective**: the probability of `h(x)` outputting every number in the set of integers in $[0, k]$ must be greater that `0`.
  In layman's terms, there must be sets of inputs `x1,x2,x3,...` such that `h(x1) == 0`, `h(x2) == 1` `h(x3) == 2`, and so on. *This is rather implied from uniformity*.

No matter what is your hash function, collisions are pretty much inevitable and we can't just **keep designing a more complex hashing function** to avoid collisions as that will just consume more computing power without an actual guarantee that the function is **really good against collisions**. So instead of avoiding collisions we **resolve them** by multiple ways.

Linear Hashing

Simplest form of hashing. Instead of inserting the item at a given slot `h(x)`, we insert the item at **the first available slot starting at `h(x)`**.

If the designated slot is already occupied during an insertion (hence a collision), meaning that `h(new_data) == h(old_data)`. Assign `new_data` to `h(old_data) + 1`, effectively avoiding the collision. If that slot is ocupied as well, keep looking at the next slot until a free slot is found.

Drawbacks:

- Both **searching** and **insertion** degenerates to $\mathcal{O}(n)$, both on worst-case and average-case scenarios.
- Deletion is **really** problematic.

`h(x)` is no longer a function that outputs the location of an item, it outputs the position where the item has the highest probability to be.

Cuckoo Hashing

Cuckoo hashing uses **two hash tables** and **two *different* hashing functions**. Conflict is resolved by moving the element from the current table to the other table, and putting the new element in lieu of it in the former hash table. Each hash function correspond to one hashing table.

Let `A` and `B` be tables of size $k + 1$, and `int ha(type x)`, `int hb(type x)` be hashing functions with range k. Now the algorithm is as follows

Insertion:

Attempt to put the `x` in hash table `A`:

- In the case of a **conflict** (if `A[ha(x)]` is not empty, call this element `existing_element1`):

    - Move `existing_element1` to `B[hb(existing_element1)]`.
    - If this slot is not empty (call this `existing_element2`):
        - Move `existing_element2` to slot `A[ha(existing_element2)]`
        - If this slot is occupied:
            - This process goes *on and on and on*.
            - Place `existing_element1` in `B[hb(existing_element1)]`
    - Place `x` in `A[ha(x)]`.

Given unoptimized premises, it is possible that the insertion process exhibits an **infinite loop**. If this happens, a failsafe may be implemented such that the table is **rehashed** such that new hashing functions are introduced and all the current items in the table will reinserted. Note that **rehashing is a computationally expensive procedure**.

Advantages of Cuckoo Hashing:

- Lookup is $\mathcal{O}(1)$ worst and average case, just need to check 2 tables.
- Insertion is $\mathcal{O}(1)$ on average case.

Disadvantages of Cuckoo Hashing:

- Insertion has an **unbounded worst case** complexity (*infinite loop*).
- Infinite loop can be fixed by rehashing after detecting that insertion has taken *too many attempts*, with several notes:
    - There is no **deterministic** way to detect whether a specific input will cause insertion to go on indefinitely.
      (further reading for the interested: *Halting Problem*)
    - To resolve this conflict, we set a fixed *max attempts constant* (that is, an arbitrarily large number) and set the dictionary to **rehash** once we have tried exceeded this number of insertion attempts. That is, to set up two **new** hash functions and **reinput** all the existing elements to new hash tables. However, **rehashing is expensive (we have to reinsert data to the hash table with unbounded worst case scenario) or impossible (when the two new hash functiont cause an infinite loop as well).**

Due to this possibility, we need to be careful with **choosing the combinations of hashing functions** as well, add a new criteria to our hash function:

- **Unique**: probability of **having a conflict in *both hash functions* is none**.
  *This is done to prevent infinite loops, but does not **prevent it entirely**. It prevents the 'trivial infinite loop', where two items in `A` and `B` exchanges places between each other.*

Nested Structures

Creates a secondary data structure in each array slot (such as a linked list or a tree), this requires additional memory space but it is the most reliable hashing method.

Suppose that `A` is our array of values, where our values are of type `type_t`.

With this method `A` is not a `type_t[]` array type but each element of `A` will be a **linked data structure**, such as a linked list or a tree, where the latter is more complicated. Each of the element of the list have same `h(x)` values.

Advantages:

- The most reliable hashing method, this will actually resolve all collisions.
- The worst case scenario of insertion and lookup will depend on the linked data structure used in the bucket.
    - For a binary tree, the insertion and lookup degenerates to $\mathcal{O}(\log n)$.
    - For a linked list, the insertion and lookup degenerates to $\mathcal{O}(n)$.
- The average case scenario will be $\mathcal{O}\left(\frac{n}{M}\right)$ or $\mathcal{O}\left(\frac{\log n}{M}\right)$, where M is the size of the array used for hashing.

Disadvantages:

- Requires overhead space.

Function Pointers

Similarly in Python, function pointers allow you to *hold functions as a variable.* This allows great abstraction as you can pass function pointers as arguments and modify how your program behaves. They are often useful to decide sorting order by making a 'key function'.

```
double (*func)(double); //pointer to a function that takes a double and outputs a double.
int (*func)(int, int); //pointer to a function that takes two ints and outputs an int.
int (*func)(void*, void*); // pointer to a function that takes two pointers of any type, and outputs an int.
```

Note that since `C` is static-typed. The function pointer needs to explicitly declare the number of arguments and their types that the function it points to may take.

Abstraction Implementations

Imagine a function that takes an `int` array `A`, and outputs an `int` array `B`. Where the `i`-th element of `B` is related with the `i`-th element of *by some function*. We can simply write:

```
void *map(int *A, int arraySize, int(*method)(int)) {
  int *B = malloc(sizeof(A));
  assert(B != NULL);
  for (int i = 0; i < arraySize; i++) {
    B[i] = method(A[i]);
  }
  return B;
}
```

Problem Solving Strategies

Generate and Test

Try values until the one that meets a specified criteria is found.

Finite Solution Space:

- Need to be done systematically, in a specific order.

Infinite Solution Space:

- Need to be done in a specific order, include a "not found" condition.

Known problems:

- Primality Test
- Finding root of an equation

Divide and Conquer

- Break the problem recursively into smaller instances.
- Solve the smaller instances, perhaps recursively.
- Combine the solutions to create solution to the original problem.

Known problems:

- Tower of Hanoi
- Quicksort
- Sums of Subsets

Monte Carlo methods

Use psuedo-random number generation to allow modelling of a specific physical problems.

Random Numbers

In C, psuedo-random numbers can be generated using the `rand()` function. `rand()` returns a random unsigned integer between `RAND_MIN` to `RAND_MAX`.

Numbers cannot be generated to true randomness in computers. Random numbers have to be seeded before they can be generated. A simple implementation:

```c
srand(time(NULL));
int i = rand();
```

The first line **seeds** the random number generator with the current system time as a seed. The second line calls a random integer between `0` and `RAD_MAX`.

`rand()` returns a different number everytime it is called, forming a sequence.

Random numbers generated with the same seed **will form the same sequence**. So if the seed is a constant, the program will generate the same sequence of numbers everytime it is ran, diminishing the value of *randomness*. Hence the seed is often that something that changes when run at a different time such as:

- The current system time.
- Amount of time between first two keystroke of the user.

Numerical Representations and Computation

Binary Numbers

All numbers are represented by **bits**, a digit of `1` or `0`. 8 **bits** make up one **byte**.

Both bytes and bits are in the form of base-2 numbers, with the only difference being the number of digits as shown above. Binary numbers can be easily converted to and from the natural base-10. Binary numbers (and decimal numbers) are read in a systematic way by multiplying the `i`-th digit and the `i`-th power of the binary.

Let $N_b(x)[i]$ be the i-th digit (**from the right!**) of a number in base b, then:

$$x = N_b(x)[0] \times b^0 + N_b(x)[1] \times b^1 + N_b(x)[2] \times b^2 + \cdots$$

For example, the number 4137 in base 10, then we have the digits $N_{10}(x) = \{7, 3, 1, 4\}$. So the number is read as (each term corresponds to i-th digit from the right):

$$4137 = \underbrace{7}_{i=0} \times 10^0 + \underbrace{3}_{i=1} \times 10^1 + \underbrace{1}_{i=2} \times 10^2 + \underbrace{4}_{i=3} \times 10^3$$

The same rule goes to base 2, the number 11011 in binary is read as:

$$(11011)_2 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 = 27$$

So the number 1011 in binary is actually 27 in decimal.

To convert decimal x to k-bit binary:

- Start with x_0, set $b = \underbrace{0000 \cdots 0}_{k-bits}$.
- If $x_0 - 2^k \geq 0$:
  - The **first digit from the left** of b is 1,
  - Set $x_1 = x_0 - 2^k$
- If $x_1 - 2^{k-1} \geq 0$:
  - The **second digit from the left of** b is 1,
  - Set $x_2 = x -_1 2^{k-1}$

  $\vdots$
- If $x_i - 2^{k-i} \geq 0$:
  - The $(i+1)$-th digit of b is now 1,
  - Set $x_{i+1} = x_i - 2^{k-i}$.

We keep on repeating this until $k = 0$, where x is simply 0 or 1. Note that if $x - 2^{k-i} \leq 0$ then we don't change the value of x for the next iteration.

Signed Integers

The way we represent integers above only works with strictly positive integers, (called **unsigned integers**). Representation of negative numbers can be done in two ways:

Sign-Magnitude (s-m):

The sig-mag (or s-m, s/m, s-mg) method changes interprets the **left-most** (the most significant bit) as the **sign**. This effectively changes the range of a number b bit number from $[0, 2^b - 1]$ to $[-2^{b-1}, 2^{b-1} - 1]$ For example, a number of 5 bits 01001 can be broken down into two parts:

$$\underbrace{0}_{\text{sign}} \quad \underbrace{1001}_{\text{mag}}$$

Where the **mag** part is just the magnitude of the number (in this case $(1001)_2$ or 4), and the **sign** part defines the sign such that:

- If `sign` is 0: the number is positive.
- If `sign` is 1: the number is negative.

In this case the number $(01001)_{2s}$ represents 4, while $(11001)_{2s}$ would represent -4.

Two's Complement:

Adding two really large integers may result in a smaller integer due to overflow. Two's complement is a representation **utilizes overflow** and allows us to **subtract two numbers by adding them**.

To convert a **decimal into two's complement**:

- **Step 1:** Convert the number's magnitude to binary as usual.
- **Step 2**: If **the number is negative**, then **negate** all the bits and add 1 to the number.
  (to negate bits: to change all $0 \to 1$ and all $1 \to 0$ in the digit)
- If adding one adds an extra bit to the number, exclude the leftmost bit from the final answer (overflow).

For instance, consider converting the number -11 to **two's complement binary form**, what we need to do:

$$-11 \to \underbrace{001011}_{\text{11 in bin}} \to \underbrace{110100}_{\text{negate bits}} \to \underbrace{110101}_{\text{add 1}}$$

Note that, the **two's complement form of a binary number is simply it's regular binary form**.

We can add two numbers by adding as follows, consider doing the simple subtraction $18 - 11$, where 18 is 010010 in base-2 and -11 is seen above. Then we can add the binary numbers:

$$\begin{aligned} &010010 \\ +&\underline{110101} \quad \text{(by overflow)} \\ =&\,000111 \end{aligned}$$

Converting 000111 to decimal gives $1 + 2 + 4 = 7$, which is in fact equal to $18 - 11$.

Note that just like **s-m** represented numbers, **the first bit being 1 indicates that the number is negative**. However, the value of the magnitude is not as trivial as **S-M** numbers.

To convert from **two's complement to decimal**:

- **If the leading bit is 1**: deduct 1 then negate all the bits. Label the number as negative.
- **Then,** convert that number to decimal as per usual to get **its magnitude**.

Binary Numbers with Fractional Parts

Numbers in base 2 **can also have fractional parts**, i.e. it can represent numbers between 0 and 1. To do so, these numbers are put to the right side of a '.' character (a **fractional point**). This is important before we move on to the next part of number representations

For instance, 1001.1101 is a valid binary number. In this case:

- $N_2[0]$ is the number to the left of the decimal point.
- The number i-digits to the left of $N_2[0]$ is $N_2[i]$.
- The number i-digits to the right of $N_2[0]$ is $N_2[-i]$.

Then, **to read a binary number** with a point is simply extending our previous method:

$$x = \cdots + N_2[-2] \times 2^{-2} + N_2[-1] \times 2^{-1} + N_2[0] \times 2^0 + N_2[1] \times 2^1 + N_2[2] \times 2^2 + \cdots$$

So the number 1001.1101 is equivalent to:

$$\begin{aligned}(1001.1101)_2 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} \\ &= 8 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} \\ &= 9 + 0.8125 = 9.8125\end{aligned}$$

Floating Point Representations

The types `float` and `double` represents *real numbers* (or more of *rational* numbers since they have limited number of digits). The difference between the two are simply the number of binary digits that is to hold them - `float` and `double` can be represented as binary too.

A floating point number in binary consists of three parts:

- **Sign** (1 bit).
  `1` indicates number is negative, `0` indicates number is positive
- **Mantissa** (n bits)
  IEEE standard: n=23 for `float` *or n=52 for* `double`
- **Exponent** (k bits), **stored as a two's complement (in this course)**.
  IEEE standard: k=8 for `float` *or k=11 for* `double`

The values of n and k are often up to the system developers.

The common convention (*IEEE Standard*) for `float` data types is to use 23 bits for the mantissa and 8 bits for the exponent. So, as an example, a digit will be represented in the computer as:

$$\underbrace{\underbrace{1}_{\text{sign}}\quad\underbrace{00111011}_{\text{exponent}}\quad\underbrace{1100000000000000000000}_{\text{mantissa}}}_{\text{total} = 32 \text{ bits} = 4 \text{ bytes}}$$

Now, recall that the `float` and `double` data types are 4 and 8 bytes in size respectively. Converting from bytes to bits, we get that **`float` and `double` data types are 32 or 64 bits in size respectively**. So, we know that depending on the data type, the total size ($1 + n + k$) add up to 8 or 16, and most importantly: its constant. So, the larger size allocated to the mantissa will cause the size of the exponent to be smaller and vice versa.

So if a systems designer **choose n to be large**, then:

- We will be able to represent more **significant digits** (more precision).

- The range of numbers that we will be able to represent will be low.

To read this number:

Let e, s be the exponent and mantissa of a number x respectively. Then convert a floating-point binary to decimal scientific we just do:

$$x_2 = \pm 0.s << e$$

$$x = \pm 0.s \times 2^e$$

Where the sign is identified by the first bit.

For example, suppose that we have a 16-bit number with exponent and mantissa sizes of 3 and 12 respectively: $1\ 011\ 110010000000$.

Then, to convert this to decimal:

- The first bit is 1, so our number is negative.
- The exponent part is 3.
- Now, the mantissa part represents the number $(0.11001)_2$ (a fractional binary).
  In decimal, this is $1/2 + 1/4 + 1/32$.
- Now we shift the digits 3 to the left (or shift the 'decimal point' to the right).
  Therefore our number in binary is: 110.01, which is $4 + 2 + 1/4 = 6.25$.

To convert decimal to binary floating:

- Shift the decimal point until all numbers are to the right of it, counting how many times it is shifted to the left. **This is your exponent**, shifting to the right means negative exponent.
- Convert the new decimal number to binary. Your answer should be of form $0.s\cdots$
- Everything except the '$0.$' is your **mantissa**. However, you will need to add appropriate number of 0s to the right of this number.
- Change the first bit according to sign as appropriate.
- **Example** (convert -7.5 to 16-bit floating point binary):
  - $-7.5 \to 0.75$ takes one shift to the left, so your exponent is 1.
  - Now we see that $0.75 = 0.5 + 0.25$, so our mantissa is 0.11 in binary.
  - Therefore so far we have $001\ 1100000000000000$.
  - Since the number is negative, append 1 to the left:
  - **Answer**: $1\ 001\ 1100000000000000$.

Numerical Processing

An algorithm that involves finding the right numeric answer to a certain problem is often needed. Such as to solve a mathematical equation.

Our computers

Limited Precision and Unwanted Behavior

Recall that our `float` and `double` types have **limited precision**. This means that two values that are actually equal can be perceived as not equal, consider the two functions:

$$f(x) = x(\sqrt{x+1} - \sqrt{x})$$

$$g(x) = \frac{x}{\sqrt{x+1} + \sqrt{x}}$$

These two quantities $f(x), g(x)$ are equal (*can be proven algebraically using conjugate multiplication*). In the computer? Not so.

Consider the ratio $r = \frac{\sqrt{x+1}}{\sqrt{x}}$, we can say that for really-really large values of x that $r \approx 1$, which indicates that $\sqrt{x+1} \approx \sqrt{x}$. Due to the **computer's really limited precision**, it can see $\sqrt{x+1} = \sqrt{x}$ for large numbers, what does this say about $f(x)$ and $g(x)$?

Now with this behavior in mind we can say that the computer sees these functions as:

$$f(x) \approx x(\sqrt{x} - \sqrt{x}) \not\approx x(\sqrt{x+1} - \sqrt{x})$$
$$g(x) \approx \frac{x}{2\sqrt{x}} \approx \frac{x}{\sqrt{x+1} + \sqrt{x}}$$

Now we can see that (for really big x), $f(x) = 0$, but $g(x) \neq 0$. The quantities $f(x)$, $g(x)$ are different values in the computer, and most importantly $f(x)$ can be interpreted as a zero, while $g(x)$ still approximates the function properly. **Be careful with subtractions**, **always use a formula that does not involve subtraction whenever possible**.

Another example to consider is the following sum:

$$S(n) = \sum_{k=1}^{n} \frac{1}{k}$$

Note that this sum is **mathematically divergent**, meaning that as n gets arbitrarily large, $S(n)$ will get arbitrarily large as well and grow without bounds. However, if we were to simulate this behavior in a computer. The results may seem different, as after a specific value of n, $\frac{1}{n}$ gets treated as if it is $= 0$. This means that we can find an n such that $S(n) = S(n+1) = S(n+2) = \cdots$, as the computer stops adding.