Notes for ECE 26400 - Advanced C Programming

Ezekiel Ulrich

October 12, 2023

Contents

Course Introduction

Continuation of a first programming course. Topics include files, structures, pointers, and the proper use of dynamic data structures. This class will be taught by Prof. Joy Xiaoqian Wang. There will be four online exams, weekly online quizzes, and 20 homework assignments. For more information, consult the syllabus here.

Tools

UNIX System: The environment we'll use in this course. No matter your machine, you can use the UNIX environment. Some common commands in UNIX-like systems are:

For a comprehensive list of UNIX commands, see Wikipedia's excellent page

- `ls` - List directory contents
- `cd` - Change directory
- `mkdir` - Make directory
- `rm` - Remove files or directories. Use -rf to recursively delete files regardless of permission
- `mv` - Move files or directories
- `diff` - Comparing two files and showing their difference. Use -w to ignore whitespace
- `cat` - Shows contents of file without opening
- `cp` - Create a copy of a file
- `[CTRL + U]` - Undoes what was last typed
- `chmod` - Change file permissions
- `chown` - Change file ownership
- `kill` - Terminate processes
- `ssh` - Secure shell remote login
- `scp` - Securely copy files between hosts
- `wget` - Download files from the web
- `find` - Search for files and directories
- `vim` - Powerful text editor

To use these, simply type them in bash. For example, ls will print the contents of your directory.

Listing 1: Using ls

```
$ ls
code-folder/    helloworld.c    homework/
```

Git: Distributed version control system. Git helps you manage, store, and collaborate on your project. The "version control" refers to how Git stores previous versions of your code, so unwanted changes can be reverted. Git is useful for when several people work on a project at a time or when you want to keep track of changes.

When using Git, you will have a staging area on your computer where you directly edit your code, a local repository (or repo) that tracks all the files associated with a project, and a remote repo to store the project. For this class, the remote repo is what's that's graded (sending TAs to check student computers took too long).

You code on your local repo and then push it to the remote repo. You can also pull updates from the remote repo to your local.

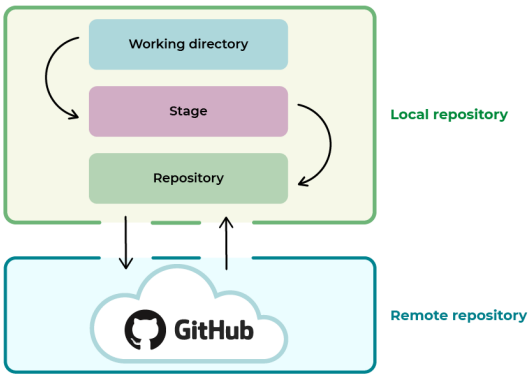

Figure 1: Layout of staging area, local repo, and remote repo

Some common Git commands are:

- `git push` - Replace what's on the remote repo with your local repo
- `git pull` - Replace your local repo with the remote repo
- `git init` - Creates a new Git repository
- `git clone` - Gets repo from specified url and copies to your machine, creating a new local repo
- `git add` - Adds file to staging area
- `git status` - Check what files in the working directory are added or committed
- `git log` - Check different versions of each project
- `git commit -m` - Moves changes from staging area to local repo. Use -m to add a message, and push to remote repo with `git push`

- `git reset` - Resets local repo to earlier version

GCC: Compiles C code to executable program. Compiling a file with gcc is simple:

Listing 2: Using gcc

```
$ gcc homework-one.c
```

Here are some useful gcc options:

- `gcc [filename] -o [output name]` - Change executable file name
- `gcc -c [filename]` - Outputs as object file
- `gcc -o [filename]` - Outputs as executable file
- `gcc -g [filename]` - Generates debug information to be used by GDB debugger.
- `gcc -Wall` - Enables all compiler's warning messages. This option should always be used, in order to generate better code.

Makefile: Allows us to specify which options should be used when gcc is called.

Listing 3: Makefile

```
GCC=gccc
CFLAGS=-std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror
EXEC = sort
TESTFLAGS = -DASCENDING

all: main.c sort.c
	$(GCC) $(CFLAGS) -o $(EXEC) main.c sort.c

# In general
target: [dependencies]
	$(GCC) $(CFLAGS) -o $(EXEC) main.c sort.c

clean:
	rm -f $(EXEC)
	rm -f *.o
```

The Makefile is invoked with the command "make" combined with a target in the terminal, like so

Listing 4: make clean

```
$ make clean
```

The test flags correspond with preprocessor directives present in your C code. For the above Makefile, perhaps we have something such as the following:

Listing 5: Conditional compilation

```
#ifdef ASCENDING
...
#endif
```

Ever been curious about the C preprocessor? It's called such because it works on your program before the compiler gets to it, replacing all instances of your `var` with the value in your `#define var` line, marking what code to ingnore between `#ifdef` and `#endif`, and `#includeing` libraries.

The compiler will only run the code in this block if we use the correct test flag when compiling.

Header file: Encapsulates formulas, function, and useful code for use in other programs. Uses ".h" extension. For compiler-included header files, include them in a preprocessor directive with triangle brackets. For user-made header files, use quotes instead.

Listing 6: Header file usage

```
#include <stdio.h>
#include "myheader.h"
```

GDB: GNU Debugger, debugger that runs on many Unix-like systems and allows you to "see" what the computer is doing as it compiles your code. Here are some useful gdb options:

- `gdb prog` - Start gdb for debug
- `b filename.c : [line no. or function name]` - Adds a break-line location specifed by line no./function name
- `info b` -
- `r` - Start the program in debugger
- `n` - Go to the next step of the function
- `s` - Step into the function
- `c` - Continue until next break point
- `list` - Show source code
- `print [variable]` - Show value of variable
- `display [variable]` - Show value of variable continuously
- `b [variable] if [condition]` - Set breakpoint when condition is met

The command to run the example file generated by -g is ./[name of example file]. The dot (.) signifies that the file is to be found in the current directory, and the slash (/) refers to a specific file.
Valgrind: Programming tool for memory debugging, memory leak detection, and profiling. Run like so:

Listing 7: valgrind

```
$ valgrind --leak-check=yes myprog arg1 arg2
```

Valgrind will run your program and try to find memory leaks and related errors. Say we have the following program:

Listing 8: valgrind example

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0; // problem 1: heap block overrun
}             // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

Valgrind will print its results like so:

Listing 9: valgrind

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 allocd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Data types and storage

Although the reader is likely familiar with data types, let us briefly recap variables for the sake of completeness. To declare a variable, we have a statement of the form

Listing 10: Variable

```
int var = 0;
```

This single line has a surprisingly rich amount of information. The `int` tells us (and the compiler) the data type, which in turn determines memory allocated, permissible operations, and what value we can assign to the variable. `var` tells us the variable's name, and `= 0` tells us its initial value. For review, the data types built into C are:

If we are curious about the size of a certain variable, we can check using the `sizeof(var)` function. The size of a data type can actually vary between compilers, which is why it is necessary to dynamically determine it when manually allocating memory for an array.

- `int`
- `double`
- `short`
- `long`
- `char`
- `void`

Users can also define or derive their own data types. Some examples of derived data types:

The `const` qualifer in C makes a variable read-only after the declaration. Later assignment of a value to the variable produces an error. We cannot make pointers to variables with the `const` qualifer unless we declare those pointers to also be `const`.

- function
- array
- pointer
- reference

When a variable is created, the compiler must assign it a location in memory. A useful analogy for this process is imagining the computer's storage as a neighborhood with a set of houses. Each house has an address, just as each variable has an address. Each house takes up a certain size, just as each variable type takes up a certain size in memory. The computer only cares about the address, but to make our code readable by humans we refer to addresses by names, analogous to calling 104 North St "Wu's house". If we create a variable "`double a = 3.2`" the computer will store an address-value pair that includes the address (where the value of the variable is) and the value (what the value of the variable is).

We may be curious where the computer stores this information. For our purposes there are two types of memory: volatile and non-volatile.
Volatile: Also called primary memory, volatile memory maintains its data only while the device is powered. Examples:

- Stack
- Heap
- Program memory

Non-volatile: does not require a continuous power supply to retain the data or program code stored in a computing device. Examples:

- USB stick
- Hard drive
- CD

So address-value pairs are stored in volatile memory, specifically the stack. The stack operates in a last-in-first-out manner, which can be visualized as an actual stack. We can put something on the top of the stack, or take something off the top of the stack. We cannot go to an arbitrary index in the stack, we may only "push" (add something to the top) or "pop" (get the top value of the stack). When a function such as `main()` is called, it often invokes another functions within itself. The way a computer keeps track of what order the code should be executed in is by pushing the *return location* onto the stack, essentially where in the program the function was called. This process repeats for any nested function calls We can visualize this on the stack, as seen in fig. 1.

function n location
...
function 2 location
function 1 location
main

Table 1: Stack

Suppose `function()` is defined as follows.

Listing 11: Functions in memory

```
void function (int a, char c) { // Args: a, c
    int i = 0; // i is a local variable
    function2(); // Pushes a new block into the
    // stack. The first line will be the return
    // location.
}
```

At this point the stack looks like fig. 2. Now `function2` would execute,

...
function2 location
`i` address and value
`c` address and value
`a` address and value
function location
main

Table 2: Stack

and once it was finished, the code would pop values from the stack until it reached the return location and resume from there. Note that `function2` does not have access to `i`, `a`, or `c` since only the topmost value on the stack can be accessed. Likewise, any variables defined in `function2` would be gone once everything was popped from the stack and we returned to `function`, so any such variables are outside the *scope* of `function`. If we want to access some calculation from `function2` outside of `function2`, we must return it. When we call a function with a return value, the computer will include a *value address* in the stack, as seen in fig. 3. Now when we call `function2`

...
function2 location
`i` address and value
`c` address and value
`a` address and value
function location and value address
main

Table 3: Stack

and it returns a value, that value will be stored in the value address specified on the bottom of the stack. Interestingly, it's possible to access the value of `a` without popping out `b` and `c`. The last-in-first-out property only applies to frames (i.e. function's frame, function2's frame, etc.). To access `a` we simply go to the address dictated by the stack pointer plus the offset corresponding to `a`. To clarify what's possible with a stack and what's not, let's consider a faulty function `swap`, which is meant to swap the value of two variables.

Listing 12: Swap

```
void swap (int a, char c) {
    int tmp = a;
    a = b;
    b = tmp;
}
int main() {
```

```
    int a = 1;
    int b = 2;
    swap(a, b);
    return 0;
}
```

If we examine the stack as this function executes, we can see why the values of `a` and `b` will actually remain unchanged. Once `swap` runs,

c address2, 1
b address2, 2
a address2, 1
return location
swap
b address, 2
a address, 1
main

Table 4: Stack

everything in the top frame will be popped off. We can see that the original `a` and `b` in `main` will be unchanged.

A useful construct is the array. The computer creates an array by storing the address of the first element (perhaps at address 100) and the address of each consecutive element in consecutive addresses (101, 102, ...). 100, 101, 102... are used only for illustration, since the computer determines what address a variable is stored in. If we want to mess around with addresses, we can use a

Pointer: a variable that stores an address. Say we have some variable `v` and we'd like to see where the computer has stored it. We can check like so:

Listing 13: Pointers

```
int main() {
    char b = 66;

    printf("value␣of␣v␣is␣%c\n", b);
    printf("address␣of␣v␣is␣%p\n", &b);

    return 0;
}
```

The & character returns the memory location of the variable to which it is prefixed. In this case it would display the address of `b` as some number in hexadecimal, perhaps `0x7ff1a990af`. We used the %p format specifier to indicate that we want to print the value of a pointer. The address will always be positive and not NULL, and will change each time the program is run as available memory changes. Let's see

another example using pointers

Listing 14: Pointers pt. 2

```
int main() {
    int x = 66;

    int *p = &x; //could also write
    //int* p or int * p

    return 0;
}
```

Let's peek inside the stack during the execution of this program.

p, 101, 100
x, 100, 6

Table 5: Stack

Here I've used the imaginary addresses 100 and 101. We see that the value of `x` is 6, while the value of `p` is the address of `x`, 100. If we want to use `p` to get the value of `x`, we can use "`*p`", which will point the computer to the address and the value stored there. If you feel like getting wacky, use `*p = 10;` to set the value of x to 10. We can even make pointers to pointers (`int** pp = &p`). To clarify, "*" can be used on pointers to access the value at the address they store, while "&" can be used on any variable to get its address. It's important to also note that you cannot mix types of pointers and types of variables. That is, something like

Listing 15: Type error

```
int main() {
    int *p;
    char c = 'c';

    p = &c; //problematic line

    return 0;
}
```

will throw an error. To ensure you've fully understood how pointers and addresses work, try to fix the `swap` program from earlier. One possible solution is

Listing 16: Corrected swap

```
void swap(int* p1, int* p2) {
    int c = *p1;
```

```
    *p1 = *p2;
    *p2 = c;
}
```

Strings and files

In C, a string is an array of characters terminated by the null terminator '\0'. For example, `char arr[] = {'I', 'U', ' ', 's', 'u', 'c', 'k', 's', '\0'}`. Some string-related functions:

- `strlen(arr)` will return the length of the string, not including the null terminator.
- `strcopy(arr1, arr2)` will copy from `arr2` to `arr1`, if there's enough space.
- `strcmp(arr1, arr2)` compares two strings. Its behavior is interesting and to understand it, we have to understand what the function is doing. Say we have the following code.

Listing 17: strcomp

```
int compare() {
    arr1 = "apple";
    arr2 = "peach";

    return strcomp(arr1, arr2);
}
```

`strcmp` will go through letter by letter and compare the ASCII values of each character. It returns a value less than 0 when the first not-matching character in arr1 has a greater ASCII value than the corresponding character in arr2. It returns a value more than 0 when the first not-matching character in arr1 has a lesser ASCII value than the corresponding character in arr2. It will return zero if they are equal. If two strings are equal up to the point at which one terminates (that is, contains a null character), the longer string is considered greater. What will it return in the above example?

- `strstr(string, substring)` finds a substring within a given string. Returns a pointer to the first char of the first occurrence of the match. That means you need to use `char*` when getting a return value.
- `strtol(str, NULL, base)` locates and returns long in `str`, in the given base (which is usually 10). It will stop searching once it hits a character, which means running it with a string like "p1000" will return 0.

The `main()` function is actually capable of handling inputs. You may see it written like so: `int main(int argc, char** argv)`. Here,

`argc` is a count of the number of arguments, and `argv` stores said arguments. The `char **` means that `argv` is an array of strings.

Now, away from strings and on to files. Opening a file in C is easy:

Listing 18: fopen

```
int main(int argc, char** argv) {
    if(argc != 2) {
        return EXIT_FAILURE;
    }

    FILE* fptr; //pointer to variable of type FILE
    fptr = fopen(argv[1], "r"); //opens to "r"ead
    if(!fptr) { //checks to see if fopen worked
        return EXIT_FAILURE;
    }

    //logic here

    fclose(fptr); //to prevent leaks
}
```

Some file-related functions:

- textttfopen opens a file in a specified mode.
- `feof` will return `true` at end of file.
- `fscanf` functions like `scanf`, but takes in file pointer as first argument. and reads from file.
- `fgetc` gets the next character from the specified file.
- `fseek` sets the file position of the stream to a given offset.
- `fprintf` functions like `printf`, but takes in file pointer as first argument.
- `fclose` closes the file. This should always be called after a successful fopen.

Dynamic memory allocation

It is often the case in coding that we need to create an array but we don't know what size it will be. A possible workaround is making a very large array (perhaps of size 10000000), but this is wasteful and it still may not be enough. Luckily, there is a way to dynamically allocate memory in C with the `malloc` (memory allocate) function.

Listing 19: malloc

```
int num;
scanf("%d", &num);

int* arr;
arr = malloc(num*sizeof(int));
if(arr == NULL) { //can't free up memory that
                  //doesn't exist
   return EXIT_FAILURE;
}

free(arr); //prevents memory leaks
```

`malloc` will return either `NULL` if it failed, or the address of the allocated memory block. Let's peek inside the memory of the computer as the following code is running.

Listing 20: malloc example

```
int num;
scanf("%d", &num); //input 3

int* arr;
arr = malloc(num*sizeof(int));
if(!arr) { //same as arr == NULL
   return EXIT_FAILURE;
}

for(int i = 0; i < num; i++) {
   arr[i] = i*2;
}

free(arr); //prevents memory leaks
           //by freeing memory in the heap
```

Static memory is allocated in the stack, while dynamic memory is allocated in the heap.

arr[2]	[address]	4
arr[1]	[address]	2
arr[0]	[address]	0
arr	[address]	[hpaddress]
num	[address]	3

Table 6: Stack

arr[2]	[address]	4
arr[1]	[address]	2
arr[0]	[hpaddress]	0

Table 7: Heap

Struct

Consider how you might represent a rectangle in C with one variable. You may choose to identify rectangles by their height and width, and to pack these attributes into a single rectangle we need a *struct*.

Listing 21: Struct

```
typedef struct { //tells compiler we
                 //are creating a new type
   int len; //length attribute
   int wid; //width attribute
} Rectangle; //name of type

//create new rectangle object, rect
Rectangle rect = {.len=5, .wid=4};

//change len value
rect.len = 1;
```

If we want to use the Rectangle struct in our code, we would define it in a `.h` file and `include` it in our program. *Derived* types like `Rectangle` cannot be used with operators like ">" or "==". To compare derived data types, define a function like `rectEquals` or `rectLargerThan` that access the attributes and compare them directly. Let's see an example.

Listing 22: changeRec

```
void changeRec(Rectangle rec) {
   rec.len += 1;
   rec.wid -= 1;
}

void main() {
   Rectangle r;
   r.len = 9;
```

```
    r. wid = 9;
    changeRec(r);
}
```

Table 8 is the stack during execution. We see that the original values

r.len	ADDR	10
r.wid	ADDR	8
changeRec	line 10	
r.len	ADDR	9
r.wid	ADDR	9
main		

Table 8: Stack

of `r` are unchanged, so if we want to alter them we'll have to pass in a pointer instead.

Listing 23: changeRec pointer

```
void changeRec(Rectangle* rec) {
    *rec.len += 1;
    *rec.wid -= 1;
}

void main() {
    Rectangle r;
    r.len = 9;
    r. wid = 9;
    changeRec(&r);
}
```

Equivalent to `*ptr.att` is `ptr->att`. Both can be used interchangeably. For example,

Listing 24: -> and malloc

```
Rectangle* r;
r = malloc(sizeof(Rectangle));
if(!r) {
    return EXIT_FAILURE;
}
r->len = 15;
r->wid = 10;
free(r);
```

When dealing with structs, there is a useful function called `fread`.

Listing 25: fread

```
size_t fread(void* ptr, size_t size,
```

```
    size_t nmemb, FILE* stream);
//reads nmemb elements of data,
//each size bytes long,
//from stream, storing them at ptr
```

`fread` can be used to read and write structs to files.

Recursion

Recursion: recusrion is where a process calls itself, like so:

Listing 26: recursion

```
int func(int n) {
    //...
    int m = func(n)
    //...
}
```

To avoid infinite recursion, every recursive function should have a *base case*.

Listing 27: recursion

```
int func(int n) {
    if(n == 0) { //base case
        return 0;
    } else {
        return {1 + func(n-1)};
    }
}

void main() {
    printf("%d\n", func(3));
}
//what does func(3) return?
//what about func(n)?
```

Table 8 is the stack during recursion.

n func	ADDR RL	0
n func	ADDR RL	1
n func	ADDR RL	2
n func	ADDR RL	3
main		

Table 9: Stack

A common example of recusion is the Tower of Hanoi, also known as the Tower of Brahma or the Lucas Tower. The tower consists of three rods with n disks of different diameters. The objective is to shift the entire stack of disks from one rod to another following these three rules :

- Only one disk can be moved at a time.
- Only the uppermost disk from one stack can be moved on to the top of another stack or an empty rod.
- Larger disks cannot be placed on the top of smaller disks.

To solve this recursively, consider the base case with two plates. To solve this case, move the top plate to the third rod, move the formerly lower plate to the second rod, and move the top plate back on top of it for a total of three steps. Say we have three plates. Then we do something similar: move the top plate to the second rod. Move the second plate to the third rod. Move the top plate to the third rod (so now we have a stack of two on the third rod). Move the lower plate to the second rod, move the top plate to the first, the middle to the second, and finally plate the top plate back on them all for a total of 7 moves. After playing around with this a bit more, we'll notice that the number of steps follows the formula $s(n) = 2s(n) + 1$, so a recursive implementation might look like:

Listing 28: Tower of Hanoi

```
int s(int n) {
    if(n == 1) { //stop criterion
        return 1;
    } else {
        return(2*s(n) + 1);
    }
}
```

We can estimate the amount of time an algortihm will take with time complexity. **Time complexity**: quantifies the amount of time taken by an algorithm to run as a function of the length of the input. For example, the total number of operations selection sort performs is $(n-1) + (n-2) + (n-3) + ... + 2 + 1 = \frac{(n+1)n}{2} = an^2 + bn + c$. For big O notation, we look at the highest power. In this case the time complexity would be $O(n^2)$. Although it's inaccurate, this tells us we should expect the time to increase roughly quadratically as the number of arguments goes up. We may want to know what sorting algorithm is the fastest: the answer varies based on the task, but it turns out that merge sort has a time complexity of $O(n \log n)$. We can see in fig. 2 that $n \log n$ grows much more slowly than n^2. Merge sort is so much faster than other algorithms because it uses the principle of transitivity.

Recursion allowed us to make a superior sorting algorithm, but it shouldn't be used everywhere. Here's a general guide of how to use recursion:

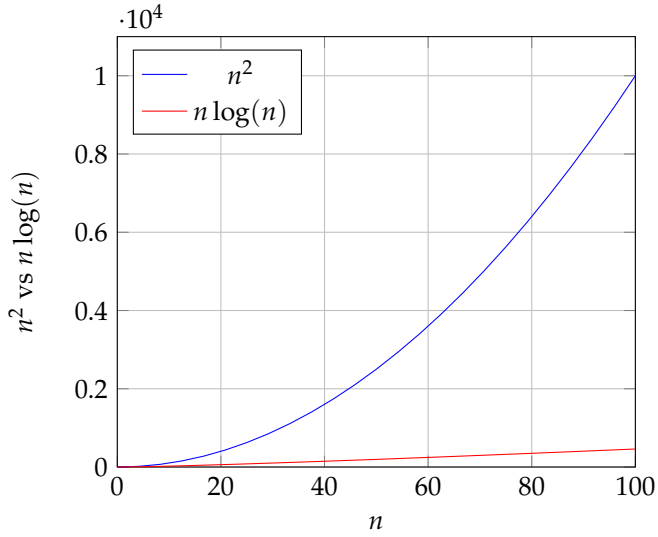

Figure 2: n^2 vs. $n \log(n)$

1. Identify the arguments of a problem.
2. Represent the solution based on the problem.
3. Derive the relation between complex cases and simpler cases.
4. Identify the simplest cases where solutions are known.

Let's try our hand at recursion with the problem of integer partitioning. Say you have some number n. How many possible ways can n be written as the sum of positive integers? For example, if $n = 3$,

$$\begin{aligned} 3 &= 1+1+1 \\ &= 1+2 \end{aligned}$$

Let's apply our steps here.

1. The positive integer to partition.
2. Call it $f(n)$.
3. We have that $f(1) = 1$. A little bit of thought will show $f(n) = 1 + \sum_{i=1}^{n-1} f(n-i)$.
4. For $n = 1$, there is 1 possible way to partition n, so $f(1) = 1$.

A real-world example of this is binary search. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the

target value is found. If the search ends with the remaining half being empty, the target is not in the array. Here's an implimentation of binary search in C:

Listing 29: Binary search

```
#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, the element was not present
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    printf("Result: %d", result);
    return 0;
}
```

Recursion is popular in many of the most effective sorting algorithms. Quicksort, one of the fastest, also uses recursion. Here's an overview of how it works:

1. Pivot Selection: Choose a pivot element from the array. The pivot

is used to partition the array into two groups.

2. Partitioning: Rearrange the elements in the array such that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side.

3. Recursion: Apply the above steps recursively to the sub-arrays on the left and right of the pivot until the base case is reached. The base case usually involves sub-arrays of size one or zero, which are already sorted by definition.

4. Combine: The sorted sub-arrays are then combined to get the final sorted array.

Linked Lists

Arrays are useful, but unfortunately are not mutable. Once their size is defined it cannot be changed afterwards. In some applications we want to have a way of storing variables whose size can be changed after initialization. A *linked list* is one way of accomplishing this.
Linked list: consists of a data element known as a node. Each node consists of two fields: one field has data, and in the second field, the node has an address that keeps a reference to the next node.

Listing 30: Linked list

```
// Defining a node
typedef struct node {
 int val;            //value of this node
 struct node* next; //address of next node
} Node;

// Creating the list
Node* head = NULL;  //NULL node means EOL
head = malloc(sizeof(Node));
head->next = NULL
head->value = 10;
```

We can do all the standard list operations with a linked list: changing the value of elements, deleting the list, and so on, and we can also add new values! If we want to add a new value, we should add it to the beginning instead of the end so our program doesn't need to traverse the entire list every time we add an element.

Listing 31: Adding to linked list

```
Node* nd = malloc(sizeof(Node));
```

```
nd->value = 15;
nd->next = head; //add the node before head
head = nd;       //make new node head
//note the order of these steps matters
```

If you don't want to repeat these steps for every node you enter, make it into functions, like the below.

Listing 32: Linked list functions

```
static Node* construct(int val) {
   Node* nd = malloc(sizeof(Node));
   nd->next = NULL;
   nd->value = val;
   return nd;
}
Node* insert(Node* head, int val) {
   Node* ptr = construct(val);
   ptr->next = head;
   return ptr;
}
void print(Node * head) {
   while (head != NULL) {
   printf ("%d ", head -> value);
   head = head -> next;
   }
}
Node* delete(Node* head, int value) {
   Node* p = head;
   if(p == NULL) {return p;}

   if ((p -> value) == val) {
      p = p->next;
      free(head);
      return p;
   }

   Node * q = p -> next;
   while ((q != NULL) && ((q -> value) != val)) {
      p = p->next;
      q = q->next;
   }

   if (q != NULL) {
   p->next = q->next; // reroute the link
```

```
        free(q); // release the memory.
        //Note that the order of "reroute" and "release" matters
        }

        return head;
    }
    void destroy(Node * head) {
        while (head != NULL) {
            Node * ptr = head -> next;
            free(head);
            head = ptr;
        }
    }
```

There are multiple types of linked lists. Another common type is the *doubly-linked list*, where each node references both the next node and the previous node, and the *circularly-linked list*, where the final node points to the first node.

Deep vs. Shallow Copy

Suppose we have code like so,

Listing 33: Struct

```
    typedef struct {
        int year;
        char* name;
    } Person

    int main() {
        Person* p = malloc(sizeof(Person));
        p->name = malloc(sizeof(char)*10);

        p->year = 2023;
        strcpy(p->name, "user);

␣␣␣␣␣␣Person␣q*␣=␣malloc(sizeof(Person));
␣␣␣␣␣␣q->name␣=␣malloc(sizeof(char)*10);

␣␣␣␣␣␣//␣How␣do␣we␣copy␣p␣to␣q?

␣␣␣␣␣␣q␣=␣p;␣//?
␣␣␣}
```