Structs

In C, we can create our own types by creating data structures. This can be done with the following syntax:

```c
typedef struct{
    char* name;
    char* phone_number;
} person;
```

This can be used to create an array with different bits of data, for example, a phonebook could be made in the following way:

```c
typedef struct{
    char* name;
    char* phone_number;
} person;

int main(){
    person people[] = {
        {"Jacob", "07123456789"},
        {"John", "07234567891"},
        {"Jane", "07345678912"}
    };
}
```

To then access these values you can use the following syntax:

```c
people[0].name;     // Jacob
people[0].number;   // 07123456789
```

phonebook.c Optimised

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

typedef struct{
  char* name;
  char* number;
} person;

int check_args(int argc, char* argv[]);
int search(int count, person people[], char* target_name);

int main(int argc, char* argv[]){
  // Check arguments
  if(check_args(argc, argv) != 0){
    return 1;
  }

  // Data Input
  int count = 3;
  person people[] = {
    {"jacob", "07123456789"},
    {"john", "07234567891"},
    {"jane", "07345678912"}
  };

  // Format input to be all lower case
  char* target_name = argv[1];
  for(int i = 0; i < strlen(target_name); i++){
    target_name[i] = tolower(target_name[i]);
  }

  // Output for search
  int index = search(count, people, target_name);
  if(index == -1){
    printf("%s not found in phonebook\n", target_name);
    return 1;
  } else {
    printf("%s: %s\n", target_name, people[index].number);
    return 0;
  }
}

int check_args(int argc, char* argv[]){
  if(argc != 2){
    printf("Usage: ./phonebook target_name\n");
    return 1;
  } else {
    return 0;
  }
}

int search(int count, person people[], char* target_name){
  for(int i = 0; i < count; i++){
    if(strcmp(people[i].name, target_name) == 0){
      return i;
    }
  }
  return -1;
}
```

```
Output:
> ./phonebook
Usage: ./phonebook target_name

> ./phonebook Jacob
jacob: 07123456789

> ./phonebook JoHn
john: 07234567891

> ./phonebook jane
jane: 07345678912

> ./phonebook jack
jack not found in phonebook
```

phonebook.c

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int check_args(int argc, char* argv[]);
int search(int count, char* names[], char* target_name);

int main(int argc, char* argv[]){
  // Check arguments
  if(check_args(argc, argv) != 0){
    return 1;
  }

  // Data Input
  int count = 3;
  char* names[] = {"jacob", "john", "jane"};
  char* numbers[] = {"07123456789", "07234567891", "07345678912"};

  // Format input to be all lower case
  char* target_name = argv[1];
  for(int i = 0; i < strlen(target_name); i++){
    target_name[i] = tolower(target_name[i]);
  }

  // Output for search
  int index = search(count, names, target_name);
  if(index == -1){
    printf("%s not found in phonebook\n", target_name);
    return 1;
  } else {
    printf("%s: %s\n", names[index], numbers[index]);
    return 0;
  }
}

int check_args(int argc, char* argv[]){
  if(argc != 2){
    printf("Usage: ./phonebook target_name\n");
    return 1;
  } else {
    return 0;
  }
}
```

```c
int search(int count, char* names[], char* target_name){
  for(int i = 0; i < count; i++){
    if(strcmp(names[i], target_name) == 0){
      return i;
    }
  }
  return -1;
}
```

```
Output:
> ./phonebook
Usage: ./phonebook target_name

> ./phonebook Jacob
jacob: 07123456789

> ./phonebook JoHn
john: 07234567891

> ./phonebook jane
jane: 07345678912

> ./phonebook jack
jack not found in phonebook
```

Sorting

Most searching algorithms rely on the fact that the data is sorted into numerical order

> Remeber that letters are in numerical order according to ASCII or Unicode

Selection Sort

My Attempt

```
Loop through all the numbers
    Loop through all the rest of the numbers
        Find the smallest number
    Swap the number at the start of the loop with the smallest number
```

```
#include <stdio.h>

void print_array(int array_length, int array[], char* message);

int main(){
    // Initalise data
    int arr[] = { 7, 2, 5, 4, 1, 6, 0, 3};
    int arr_len = 8;

    // Print starting state of array
    print_array(arr_len, arr, "Starting Array: ");

    // Sort
    for(int i = 0; i < arr_len, i++){
        int s_val = arr[i];
        int s_index = i;
        for(int j = i; j < arr_len; j++){
            if(arr[j] < s_val){
                s_val = arr[j];
                s_index = j;
            }
        }

        // * Could do a XOR swap however, this is easier to learn from.
        int temp = arr[i];
        arr[i] = s_val;
        arr[s_index] = temp;
    }

    // Print ending state of array
    print_array(arr_len, arr, "Final Array   : ");
}

void print_array(int array_length, int array[], char* message){
    printf("%s", message);
    for(int i = 0; i < array_length, i++){
        printf("%i ", array[i]);
    }
    printf("\n");
}
```

```
Output:
> ./selection_sort
Starting Array: 7 2 5 4 1 6 0 3
Final Array   : 0 1 2 3 4 5 6 7
```

CS50X Solution

```
For i from 0 to n-1
    Find smallest number between numbers[i] and numbers[n-1]
    Swap smallest number with numbers[i]
```

Time Complexity

Because of the double `for` loop, selection sort is $O(n^2)$, $\Omega(n^2)$ and, therefore, $\Theta(n^2)$.

The $O(n^2)$ can be figured out by the following:

$$(n-1)+(n-2)+(n-3)+...+1$$

$$n(n-1)/2$$

$$(n^2-n)/2$$

$$(n^2)/2-n/2$$

In big O notation, we only care about the thing in the equation, which is the n^2 portion of this, therefore the algorithm is considered $O(n^2)$.

The reason why I say it is $\Omega(n^2)$, and thus $\Theta(n^2)$, is because the algorithm has no concept to check if the list is already sorted, this could be improved with a flag and breaking out early, potentially making the algorithm $\Omega(n)$ as it will only need to check through the list once.

Bubble Sort

My Attempt

```
Loop through all bumbers
```

```
    Loop from 0 to end
        If value is bigger than value next to it
            Swap value with value next to it
```

```c
#include <stdio.h>

void print_array(int array_length, int array[], char* message);

int main(){
    // Initalise data
    int arr[] = { 7, 2, 5, 4, 1, 6, 0, 3};
    int arr_len = 8;

    // Print starting state of array
    print_array(arr_len, arr, "Starting Array: ");

    // Sort
    for(int i = 0; i < arr_len; i++){
        for(int j = 0; j < arr_len - i - 1; j++){
            if(arr[j] > arr[j + 1]){
                int temp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = temp;
            }
        }
    }

    // Print ending state of array
    print_array(arr_len, arr, "Final Array    : ");
}

void print_array(int array_length, int array[], char* message){
    printf("%s", message);
    for(int i = 0; i < array_length, i++){
        printf("%i ", array[i]);
    }
    printf("\n");
}
```

CS50X Solution

v1

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
```

v2

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
    If no swaps
        Quit
```

Time Complexity

This again uses a double `for` loop, therefore it is $O(n^2)$. As well as this, it also doesn't implement a check for whether the array is already sorted, therefore it is $\Omega(n^2)$ and therefore $\Theta(n^2)$.

To improve upon this algorithm, we could have a flag that checks if any swaps were completed, if not the loop breaks. This means that the time complexity would now be $\Omega(n)$.

Merge Sort

CS50X Solution

Psuedo Code

```
If only one number
    Quit
Else
    Sort left half of numbers
    Sort right half of numbers
    Merge numbers together
```

Use this to understand how merge search works.

Time complexity Because it has to do each layer once, it does this $\log_2(n)$ times, however each layer has to have a check each time n, therefore the total time complexity is $O(n\log(n))$. The Ω time complexity is also $\Omega(n\log(n))$ and therefore bubble and selection sort can outperform this algorithm if the data is already sorted. The total time complexity is $\Theta(n\log(n))$

Searching

Array

An array is a contiguous (back-to-back) data structure of the same type:

```c
int numbers[] = {1, 5, 15, 20, 50, 100, 500};
```

Searching Algorithms

Searching algorithms take an array as input, and attempt to give a `boolean` as to whether the value you are searching for is in the array.

Linear Search

When the data isn't sorted, you brute-force by going through all the values in the array until you find it.

```c
#include <stdbool.h>
#include <stdio.h>

bool linear_search(int array_length, int* array, int target);

int main(){
  int example_array_length = 7;
  int example_array[] = {1, 5, 15, 20, 50, 100, 500};
  int example_target = 20;

  bool is_found = linear_search(example_array_length, example_array, example_target);
  printf("Is %i in array? %s\n", example_target, is_found ? "It is found!" : "It wasn't foun
}

bool linear_search(int array_length, int* array, int target){
  for(int i = 0; i < array_length; i++){
    if(array[i] == target){
        return true;
    }
  }
  return false;
}
```

Binary Search

When the data is sorted, you can use a *"divide and conquer"* technique where you check the middle, if the target is larger then take the right side of the array and try again, if the target is smaller then take the left side and do it again, else you have the value there or the value doesn't exist.

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

bool binary_search(int array_length, int* array, int target);

int main(){
  int example_array_length = 7;
  int example_array[] = {1, 5, 15, 20, 50, 100, 500};
  int example_target = 5;

  bool is_found = binary_search(example_array_length, example_array, example_target);
  printf("Is %i in array? %s\n", example_target, is_found ? "It is found!" : "It wasn't fou
}

bool binary_search(int array_length, int* array, int target){
  int start = 0;
  int end = array_length - 1;
  int middle;

  while(end - start > 0){
    middle = ((end - start) / 2) + start;
    if(target == array[middle]){
      return true;
    } else if(target > array[middle]){
      start = middle + 1;
    } else if(target < array[middle]){
      end = middle - 1;
    }
  }

  return false;
}
```

search.c

linear_search.c

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

int check_argc(int argc);
int check_argv(char* arg);
int linear_search(int datac, int datav[], int target);

int main(int argc, char* argv[]){
  if(check_argc(argc) == 1){
    return 1;
  } else if(check_argv(argv[1]) == 1){
    return 1;
  }

  int datac = 7;
  int datav[] = {20, 500, 10, 5, 100, 1, 50};
  int target = atoi(argv[1]);
  int index = linear_search(datac, datav, target);
  if(index == -1){
    printf("Target: %i not found\n", target);
    return 1;
  } else {
    printf("Target: %i found at index: %i\n", target, index);
    return 0;
  }
}

int check_argc(int argc){
  if(argc != 2){
    printf("Usage: ./linear_search target\n");
    return 1;
  } else {
    return 0;
  }
}

int check_argv(char* arg){
  int i = 0;
  while(arg[i] != '\0'){
    if(!isdigit(arg[i])){
      printf("Target should only be digits.\n");
      return 1;
    }
    i++;
  }
  return 0;
}

int linear_search(int datac, int datav[], int target){
  for(int i = 0; i < datac; i++){
    if(datav[i] == target){
      return i;
    }
  }
  return -1;
}
```

```
Output:
> ./linear_search
Usage: ./linear_search target

> ./linear_search abc
Target should only be digits.

> ./linear_search 123
Target: 123 not found

> ./linear_search 50
Target: 50 found at index: 6
```

linear_search_string

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

int check_argc(int argc);
int linear_search(int datac, char* datav[], char* target);

int main(int argc, char* argv[]){
  if(check_argc(argc) == 1){
    return 1;
  }

  int datac = 6;
  char* datav[] = {"battleship", "boot", "cannon", "iron", "thimble", "top hat"};
  char* target = argv[1];
  int index = linear_search(datac, datav, target);
  if(index == -1){
    printf("Target: %s not found\n", target);
    return 1;
  } else {
    printf("Target: %s found at index: %i\n", target, index);
    return 0;
  }
}

int check_argc(int argc){
  if(argc != 2){
    printf("Usage: ./linear_search target\n");
    return 1;
  } else {
    return 0;
  }
}

int linear_search(int datac, char* datav[], char* target){
  for(int i = 0; i < datac; i++){
    if(strcmp(datav[i], target) == 0){
      return i;
    }
  }
  return -1;
}
```

```
Output
> ./linear_search_string
Usage: ./linear_search target

> ./linear_search_string battleship
Target: battleship found at index: 0

> ./linear_search_string "top hat"
Target: top hat found at index: 5
```

Recurssion

Recurssion is when a function calls itself and is an alternative to a iterative approach by using loops. For example

```
#include <stdio.h>

void recurssive(int index, int max);
void iterative(int start, int max);

int main(){
    int start = 0;
    int max = 5;

    recurssive(start, max);
    printf("------------\n");
    iterative(start, max);
}

void recurssive(int index, int max){
    printf("Hello, world\n");
    if(index < max - 1){
        recurssive(index + 1, max);
    }
}

void iterative(int start, int max){
    for(int i = start; i < max; i++){
        printf("Hello, world\n");
    }
}
```

Mario Recurssive

```
#include <stdio.h>

void draw(int n);

int main(){
    int height = 4;
    draw(height);
}

void draw(int n){
    if(n > 1){
        draw(n - 1);
    }

    for(int i = 0; i < n; i++){
        printf("#");
    }
    printf("\n");
}
```

Running Time

Big O Notation

Big O notation removes the need for knowing the exact measurments of the timing of an algorithm, rather just the shape it creates, i.e. $O(n/2)$ and $O(n)$ are the same as $O(n)$.

"O" is described as *"the worst case scenario"*, so for example with linear search, you could get it on the first try by luck, however the worst case is that you will have to search through all n points, therefore it is $O(n)$.

$O(n^2)$, Quadratic Time.

$O(n^2)$ is typically when a double `for` loop happens. This means that every data point is relating to every other data point, i.e.

```c
#include <stdio.h>

int main(){
    int arr[] = {1, 2, 3, 4};
    int arr_len = 4;

    for(int i = 0; i < arr_len; i++){
        for(int j = 0; j < arr_len; j++){
            printf("%i + %i = %i\n", arr[i], arr[j], arr[i] + arr[j]);
        }
    }
}
```

```
OUTPUT:
1 + 1 = 2
1 + 2 = 3
1 + 3 = 4
1 + 4 = 5
2 + 1 = 3
2 + 2 = 4
2 + 3 = 5
2 + 4 = 6
3 + 1 = 4
3 + 2 = 5
3 + 3 = 6
3 + 4 = 7
4 + 1 = 5
4 + 2 = 6
4 + 3 = 7
4 + 4 = 8
```

This therefore ran 16 times with 4 data points: $4^2 = 16$.

$O(n)$, Linear Time.

$O(n)$ is linear, the amount of processing time to data is proportional, i.e.

```
#include <stdio.h>

int lin_search(int arr_len, int* arr, int target);

int main(){
    int arr[] = {1, 2, 3, 4};
    int arr_len = 4;
    int target = 4;

    int lin_search_result = lin_search(arr_len, arr, target);

    if(lin_search_result == -1){
        printf("Target: %i not found.\n", target);
    } else {
        printf("Target: %i found at index %i\n", target, lin_search_result);
    }
}

int lin_search(int arr_len, int* arr, int target){
    for(int i = 0; i < arr_len; i++){
        if(arr[i] == target){
            return i;
        }
    }

    return -1;
}
```

```
OUTPUT:
Target: 4 found at index 3
```

The worst case for this would've been looking through the entire dataset, therefore $n = 4$, $4 = 4$.

$O(\log(n))$, Logarithmic Time.

This means that the growth of time dependent on data points decreases the more you give, giving you a logarithmic shape. It typically uses the *"divide-and-conquer"* technique, i.e. with binary search:

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

```c
int bin_search(int arr_len, int* arr, int target);

int main(int argc, char* argv[]){
    int arr[] = {1, 2, 3, 4};
    int arr_len = 4;
    int target = 3;

    int bin_search_result = bin_search(arr_len, arr, target);
    if(bin_search_result == -1){
        printf("Target: %i not found.\n", target);
    } else{
        printf("Target: %i found at index %i\n", target, bin_search_result);
    }
}

int bin_search(int arr_len, int* arr, int target){
    int start = 0;
    int end = arr_len - 1;

    while(start <= end){
        int middle = start + (end - start) / 2;
        if(target == arr[middle]){
              return middle;
        } else if(target > arr[middle]){
              start = middle + 1;
        } else if(target < arr[middle]){
              end = middle;
        }
    }

    return -1;
}
```

```
OUTPUT:
Target: 3 found at index 2
```

This means that through each iteration, the amount of data to process halves, meaning that we have a logarithmic shape.

$O(1)$, Constant Time.

This means that the algorithm is not dependent on the amount of data, and is always a constant time, i.e.

```c
#include <stdio.h>

int main(){
    int arr[] = {1, 2, 3, 4};
    int arr_len = 4;

    printf("The first item is: $i\n", arr[0]);
}
```

```
OUTPUT:
The first item is 1
```

This would always return the first item in the array, therefore if there was 100 data points in the array, it would still take just as long to return the first item, therfore the time is always going to be the same.

Ω Notation

While Big O notation refers to the worst case scenario, Ω refers to the best case scenario of an algorithm.

Most searching algorithms come under $\Omega(1)$ because you might get lucky and get it on your first try, the most famous / ridiculous of these algorithms would be *"Bogo sort"* where a random permutation of the dataset is made everytime, eventually it could be the sorted dataset, however best case scenario it could be done on it's first attempt, therefore making it $\Omega(1)$.

Θ Notation

Θ Notation is when Ω and O are the same. For example, in this algorithm the best case scenario is n, as well as the worst case scenario as it always has to go through all characters. This therefore means that this is $\Theta(n)$

```
#include <stdio.h>

int main(){
    char str[] = "Hello, world";
    int str_len = 0;

    while(str[str_len] != '\0'){
        str_len++;
    }

    printf("The length of \"%s\" is %i\n", str, str_len);
}
```

Shapes of Algorithms

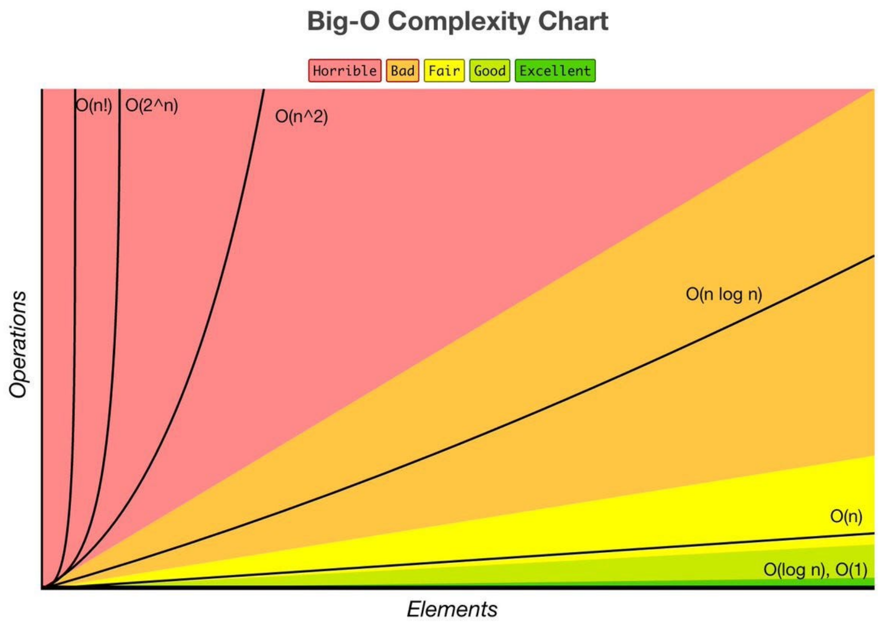

Figure 1: Shapes of Algorithms