Tutorial 7

- Arrays.
- Pointer arithmetic.
- Efficiency.

Arrays

They can be used to store a **fixed number** of elements of the **same type**.

Example of array syntax:

```
int my_array[3] = { 1, 2, 3 };
int x = my_array[0]; // x = 1
```

Array Initialization

There are several ways to define an array:

```
int a[3]; // array is not initialized, but it's defined

int b[3] = { 1, 2, 3 }; // array is initialized

int d[3] = {0}; // array of length 3, filled with zeros

int e[8] = { 7, 4, 1 };
// {7, 4, 1, 0, 0, 0, 0, 0}
```

Array Exercise

```c
// reverse_array(arr, len) reverses the contents of arr
// requires: arr is an array with length (at least) len
// effects:  modifies arr
void reverse_array(int arr[], int len);
```

Pointer Arithmetic

Certain arithmetic operations can be performed on pointers. An integer can be **added or subtracted** to a pointer, and pointers of the same type can be **subtracted** from one another.

```
int a[10];
int *p = a;      // a is a pointer to first element
int *q = &a[9]; // address of 10th element
q = a + 9;       // equivalent
a[2] = q - p;    // set the value of 3rd element as 9
q = p + 1;       // now q == &a[1]
```

Addition of pointers is not allowed.

Array Exercise 2

Write Reverse again, now using pointer arithmetic

(hint, this code will be essentially identical to reverse)

```
// reverse_array(arr, len) reverses the contents of arr
// requires: arr is an array with length (at least) len
// effects:  modifies arr
void reverse_array(int *arr, int len);
```

The syntax `a[i]` is shorthand for the equivalent expression `*(a+i)`.

Efficiency

- When looking at a function, it is often useful to understand its running time.
- To do this, we compute the running time as a function of the input size, and use Big O notation to simplify the computation.

In this course, you will see the following running times:

$$O(1)\ \ O(\log n)\ \ O(n)\ \ O(n \log n)\ \ O(n^2)\ \ O(n^3)\ \ O(2^n)$$

Recursive Functions

Recall the steps for a recursive funtion:

1. Identify the order of the function *excluding* any recursion
2. Determine the size of the input for the next recursive call(s)
3. Write the full *recurrence relation* (combine step 1 & 2)
4. Look up the closed-form solution in a table

Recurrence Relations

$T(n) = O(1) + T(n - k_1)$	$= O(n)$
$T(n) = O(n) + T(n - k_1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n - k_1)$	$= O(n^3)$
$T(n) = O(1) + T(\frac{n}{k_2})$	$= O(\log n)$
$T(n) = O(1) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n)$
$T(n) = O(n) + k_2 \cdot T(\frac{n}{k_2})$	$= O(n \log n)$
$T(n) = O(1) + T(n - k_1) + T(n - k_1')$	$= O(2^n)$

where $k_1, k_1' \geq 1$ and $k_2 > 1$

Exercise: Identifying Complexity of the Factorial Function

What is the running time of the following function?

```
// requires: n >= 0
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return (n * (fact(n - 1)));
  }
}
```

Exercise: Complexity Calculation

Consider an array of n integers. Suppose you want to know whether m particular integers exist in this array.

What is the time complexity if you sort the array using **merge sort** and search the array m times using **binary search**?

Exercise: Sort and Search

Write the following C program using merge_sort.h and bsearch.h:

1. It reads integers until it encounters 0 and puts them in an array.
2. Afterwards, it reads more integers until EOF, and prints "yes" or "no" depending on whether they are in the array.

You must use the **merge sort** module to sort the array, and use the **binary search** module to search the array.