Sorting

Bubble Sort

- Locally sorts pair of items in the array
- The pair will be sorted if their order does not meet expectations
- The large values "bubble up" to the end of the array
- Visualize Bubble Sort

Bubble Sort – Outer Loop

```c
void bubble_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        bool swapped = false;
        for (size_t j = 0; j < size - i - 1; ++j) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

- Outer loop condition is crucial
    - makes sure in the inner loop where pairs of elements are compared, it stops short of the last element

Bubble Sort – Inner Loop

```
void bubble_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        bool swapped = false;
        for (size_t j = 0; j < size - i - 1; ++j) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

- As the loop progresses, the elements towards the end of the array are sorted

Bubble Sort – Swapping

```c
void bubble_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        bool swapped = false;
        for (size_t j = 0; j < size - i - 1; ++j) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

```c
void swap(int *value1, int *value2) {
    int temp = *value1;
    *value1 = *value2;
    *value2 = temp;
}
```

Bubble Sort - Optimization

```
void bubble_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        bool swapped = false;
        for (size_t j = 0; j < size - i - 1; ++j) {
            if (array[j] > array[j + 1]) {
                swap(&array[j], &array[j + 1]);
                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

- If we didn't swap any values
  - everything is already sorted
  - we can break (quit) the loop early.

Selection Sort

- Selection Sort finds the smallest values in the array
- It then checks to make sure that there are no smaller values in the rest of the array
- When the smallest value is found, it is placed at the beginning of the array
- This process repeats moving the next smallest element in the array by one each iteration making the front of the array become a sorted portion that expands each iteration
- Visualize Selection Sort

Selection Sort- Outer Loop

```
void selection_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        size_t minimum_value_index = i;
        for (size_t j = i + 1; j < size; ++j) {
            if (array[j] < array[minimum_value_index]) {
                minimum_value_index = j;
            }
        }

        swap(&array[minimum_value_index], &array[i]);
    }
}
```

- Similar to Bubble Sort

Selection Sort- Outer Loop

```c
void selection_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        size_t minimum_value_index = i;
        for (size_t j = i + 1; j < size; ++j) {
            if (array[j] < array[minimum_value_index]) {
                minimum_value_index = j;
            }
        }

        swap(&array[minimum_value_index], &array[i]);
    }
}
```

- We are concerned with the index, not the minimum element itself
- Important for when we'll make the swap
- In bubble sort we were locally swapping adjacent elements

Selection Sort- Inner Loop

```
void selection_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        size_t minimum_value_index = i;
        for (size_t j = i + 1; j < size; ++j) {
            if (array[j] < array[minimum_value_index]) {
                minimum_value_index = j;
            }
        }

        swap(&array[minimum_value_index], &array[i]);
    }
}
```

- j starts at i + 1
  - The aim is to find from the elements after i, a minimum value witch which to swap the value at position i
  - This will swap the array up until the i^{th} index

Selection Sort- Inner Loop

```c
void selection_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        size_t minimum_value_index = i;
        for (size_t j = i + 1; j < size; ++j) {
            if (array[j] < array[minimum_value_index]) {
                minimum_value_index = j;
            }
        }

        swap(&array[minimum_value_index], &array[i]);
    }
}
```

- Finding out the index of the minimum element

Selection Sort- Outer Loop

- Making the swap

```
void selection_sort(int *array, size_t size) {

    for (size_t i = 0; i < size - 1; ++i) {
        size_t minimum_value_index = i;
        for (size_t j = i + 1; j < size; ++j) {
            if (array[j] < array[minimum_value_index]) {
                minimum_value_index = j;
            }
        }

        swap(&array[minimum_value_index], &array[i]);
    }
}
```

Insertion Sort

- Insertion Sort starts by assuming the first element is already sorted.
- The next element is then compared to that sorted value.
- If the next element is greater than the "sorted element" it stays where it is
- If it happens to be smaller, the sorted values are all shifted over to the right to make room for the value to be "inserted" in the correct spot
- Visualize Insertion Sort

Insertion Sort – Outer Loop

```
void insertion_sort(int *array, size_t size) {

    for (size_t i = 1; i < size; ++i)
    {
        int value = array[i];
        size_t j = i - 1;

        while (j >= 0 && array[j] > value) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = value;
    }
}
```

- Insertion Sort starts by assuming the first element is already sorted.
- So, the outer loop starts at i = 1
- Element at array[i] is the first element in the ‘unsorted’ part of the array
- We store array[i] in *value*

Insertion Sort – Outer Loop

```c
void insertion_sort(int *array, size_t size) {

    for (size_t i = 1; i < size; ++i)
    {
        int value = array[i];
        size_t j = i - 1;

        while (j >= 0 && array[j] > value) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = value;
    }
}
```

- The next element is then compared to that sorted value.
  - j is initialized at i – 1
- A while loop is used since multiple conditions are sometimes harder to read in a for loop
- Every element from j = i – 1 till

  j >=0 is compared against element at the i^{th} position which we're trying to insert
- If the elements being compared are greater than *value* we shift them to the right until we find a position where *value* can be *inserted*

Quick Sort

- qsort (Quick Sort) is a more efficient sort than what we have implemented
- We need to supply
  - the array
  - the size of the array
  - the size of the elements the array holds
  - a function to compare the values to know which one is "bigger" or "smaller".

```
qsort(array, 10, sizeof(int), compare);
```

compare()

```
int compare(const void *left, const void *right) {
    return *(int *)left - *(int *)right;
}
```

- We are passing a function to another function
- Functions have their own place in memory
- The name of the function is its address in memory
- Parameters: two void pointers
  - void pointers cannot be dereferenced
  - Need to typecast