Week 7 Lecture 2

Arrays, Sorting, Median

Objectives

- Understand how arrays work
  - Example: summing an array
  - Example: sorting an array
  - Example: calculating the median of a list of numbers
  - Example: searching for a number

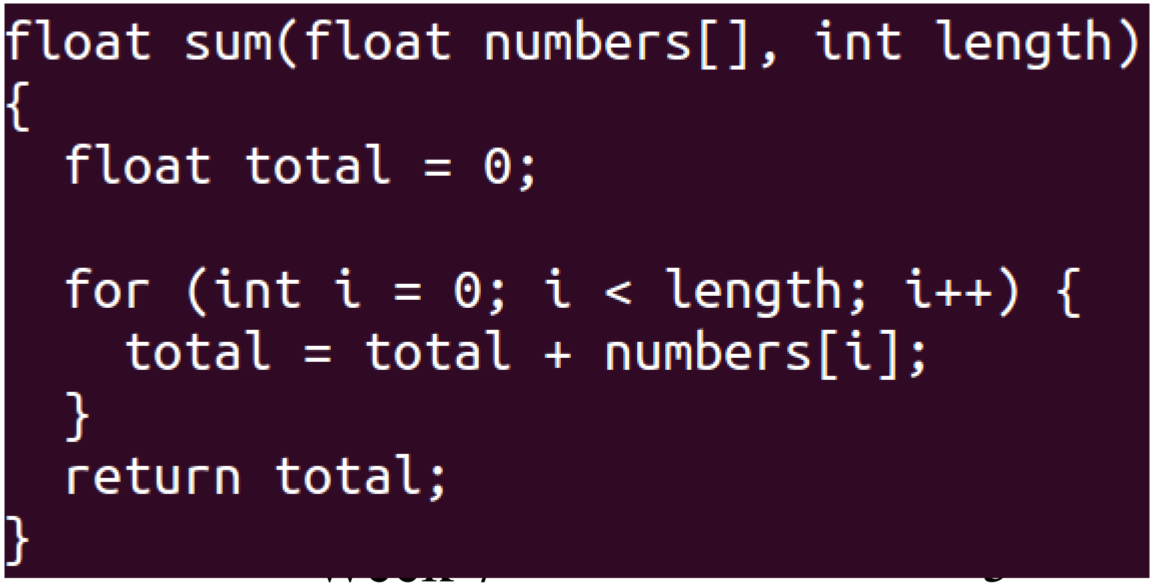
Arrays

- Arrays are a sequence of identical data items.
  - e.g. int numbers[80];
- They are accessed through an integer index.
  - e.g. number[5] = 35;
- String are an example of arrays.

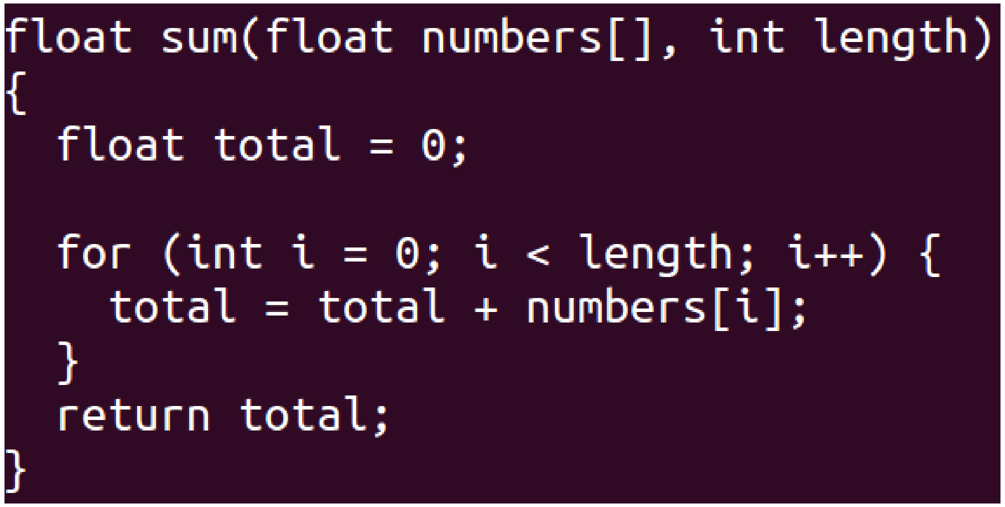
Sum

Define Sum

- Array parameter: float numbers[]
  - Need to know length of array: int length
- Add each element of the array to the total.
- Return total



```
float sum(float numbers[], int length)
{
  float total = 0;

  for (int i = 0; i < length; i++) {
    total = total + numbers[i];
  }
  return total;
}
```



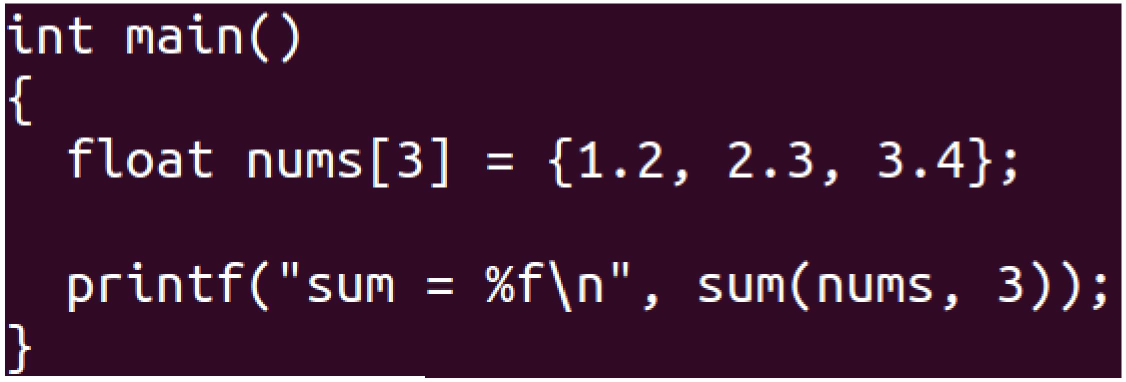
```
float sum(float numbers[], int length)
{
  float total = 0;

  for (int i = 0; i < length; i++) {
    total = total + numbers[i];
  }
  return total;
}
```

Sum

- The variable total is called an accumulator
  - Each time through the loop it accumulates the total so far
  - When the loop terminates all of the elements of the array have been added to the accumulator
- To ensure you total the entire array the loop must go through the entire array
  - Starts a 0; terminates when it has added “length” numbers.

Call Sum

- Declare and initialize array
  - float nums[3] = {1.2, 2.3, 3.4}
- Call function
  - sum(nums, 3)



```
int main()
{
  float nums[3] = {1.2, 2.3, 3.4};

  printf("sum = %f\n", sum(nums, 3));
}
```

Sort

Sorting

- Conceptually simple; computationally hard.
- An array, a_0 .. a_n, is sorted if for each a_i, $a_i \leq a_{i+1}$
- It is computationally difficulty because you have to look at all pairs.
  – Naive implementation are usually $O(n^2)$.

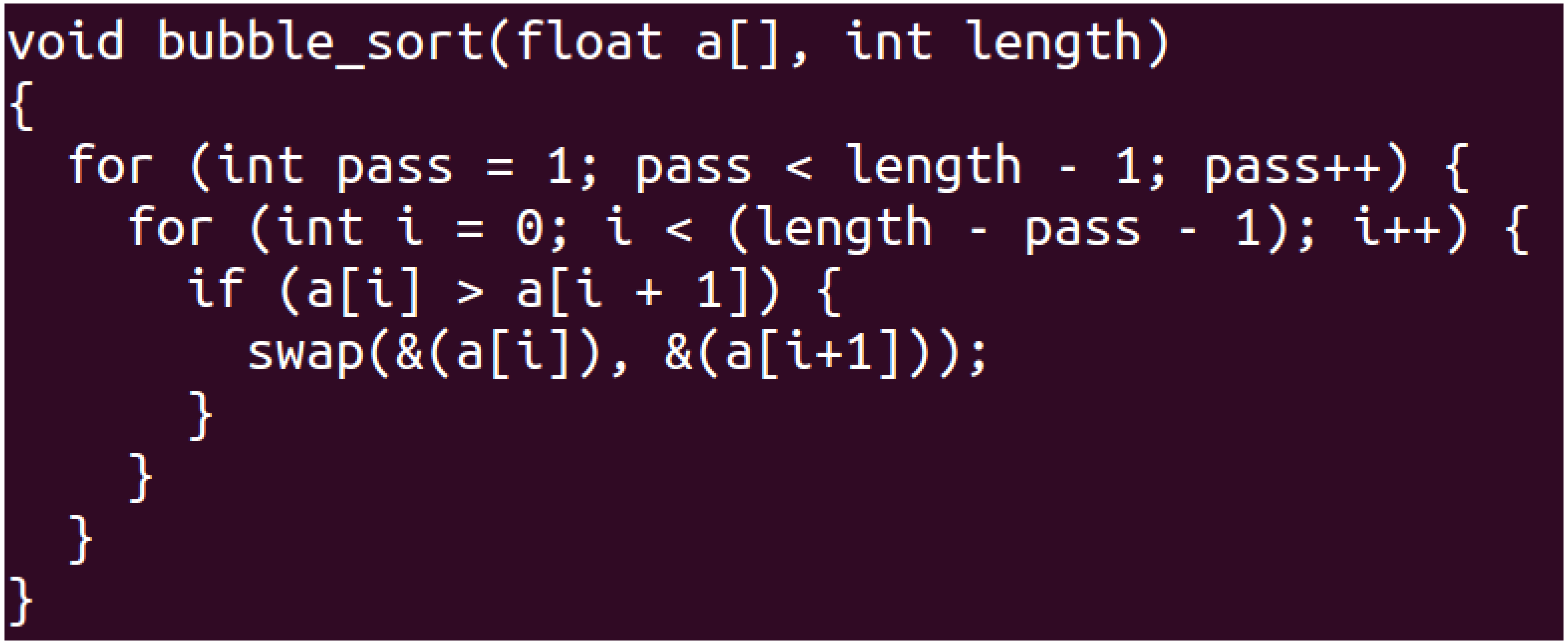
Sorting data by Using Bubble Sort

- Bubble sort algorithm:
  - Is one of the simplest sorting algorithms
  - Works by repeatedly scanning through the list, comparing adjacent elements, and swapping them if they are in the wrong order

Bubble Sort Algorithm

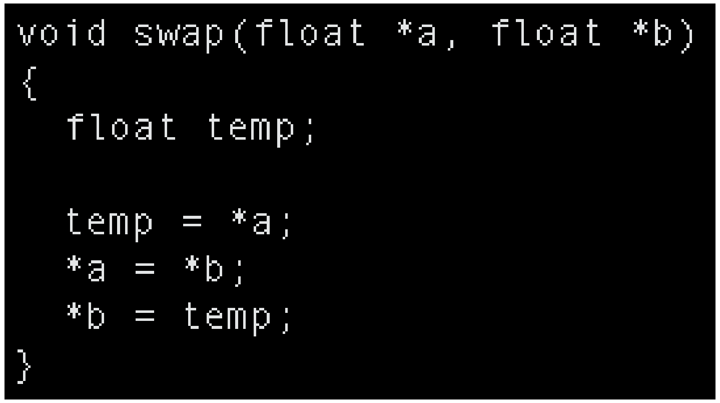
```
Algorithm Bubble ( type A[], int length )
  {
   1. for ( pass = 1 to length-1 )
          1.1 for ( j= 0 to length – 1 – pass )
                    1.1.1   If ( A [j] > A [j+1] ) {
                              1.1.1.1   t = A [j]
                              1.1.1.2   A [j] = A [j+1]
                              1.1.1.3   A [j+1] = t
                    }
   }
```

Implementing Bubble Sort

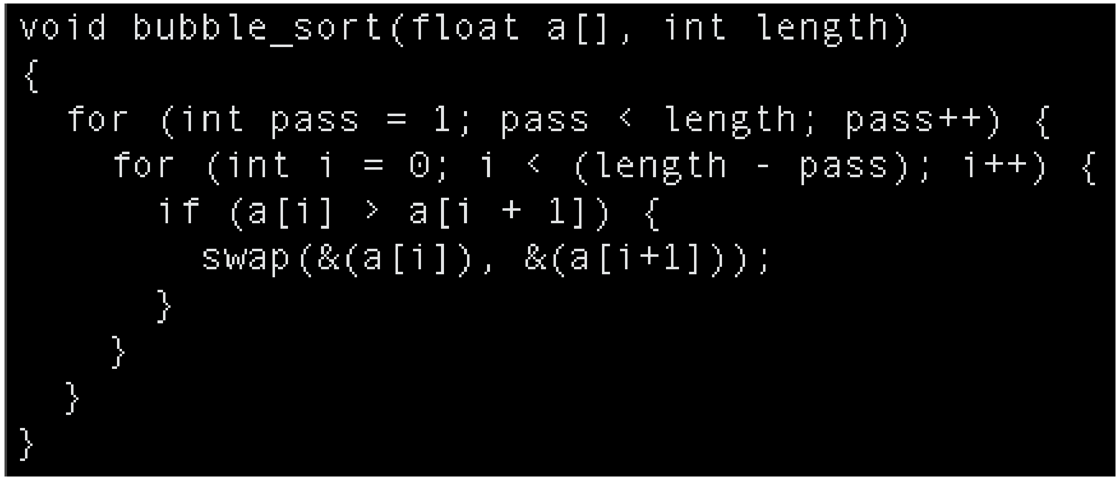


```
void bubble_sort(float a[], int length)
{
  for (int pass = 1; pass < length - 1; pass++) {
    for (int i = 0; i < (length - pass - 1); i++) {
      if (a[i] > a[i + 1]) {
        swap(&(a[i]), &(a[i+1]));
      }
    }
  }
}
```

Swap Implementation

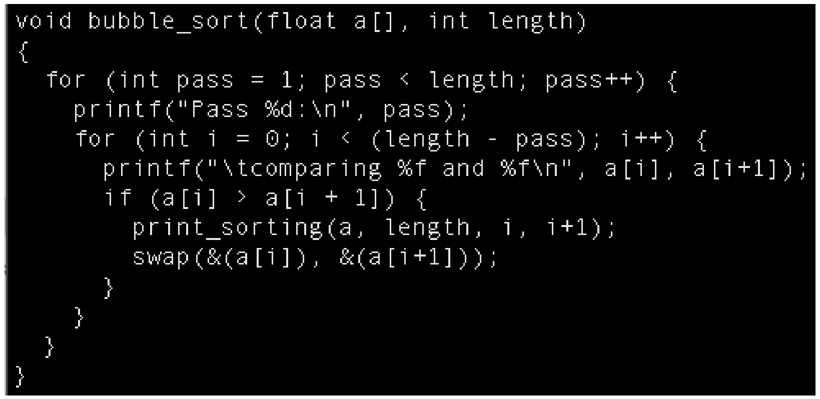


```
void swap(float *a, float *b)
{
  float temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

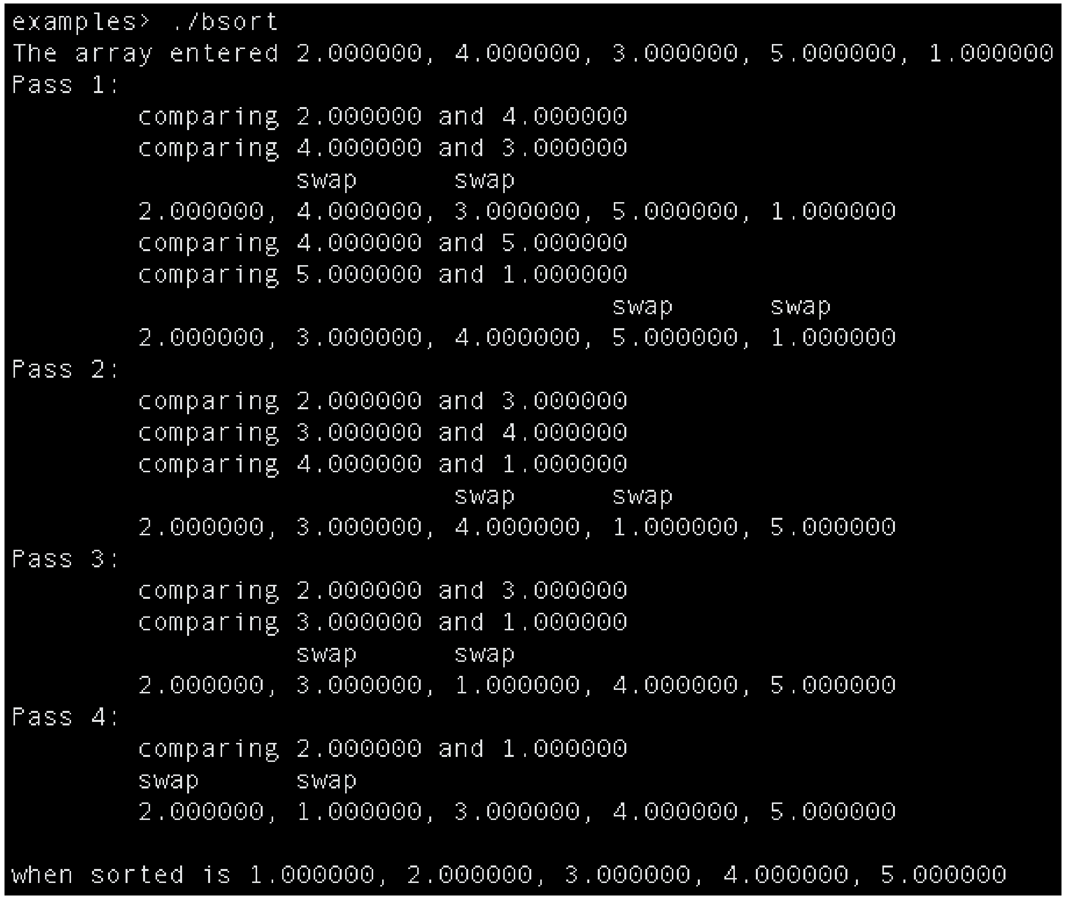


```
void bubble_sort(float a[], int length)
{
  for (int pass = 1; pass < length; pass++) {
    for (int i = 0; i < (length - pass); i++) {
      if (a[i] > a[i + 1]) {
        swap(&(a[i]), &(a[i+1]));
      }
    }
  }
}
```

Bubble Sort Application

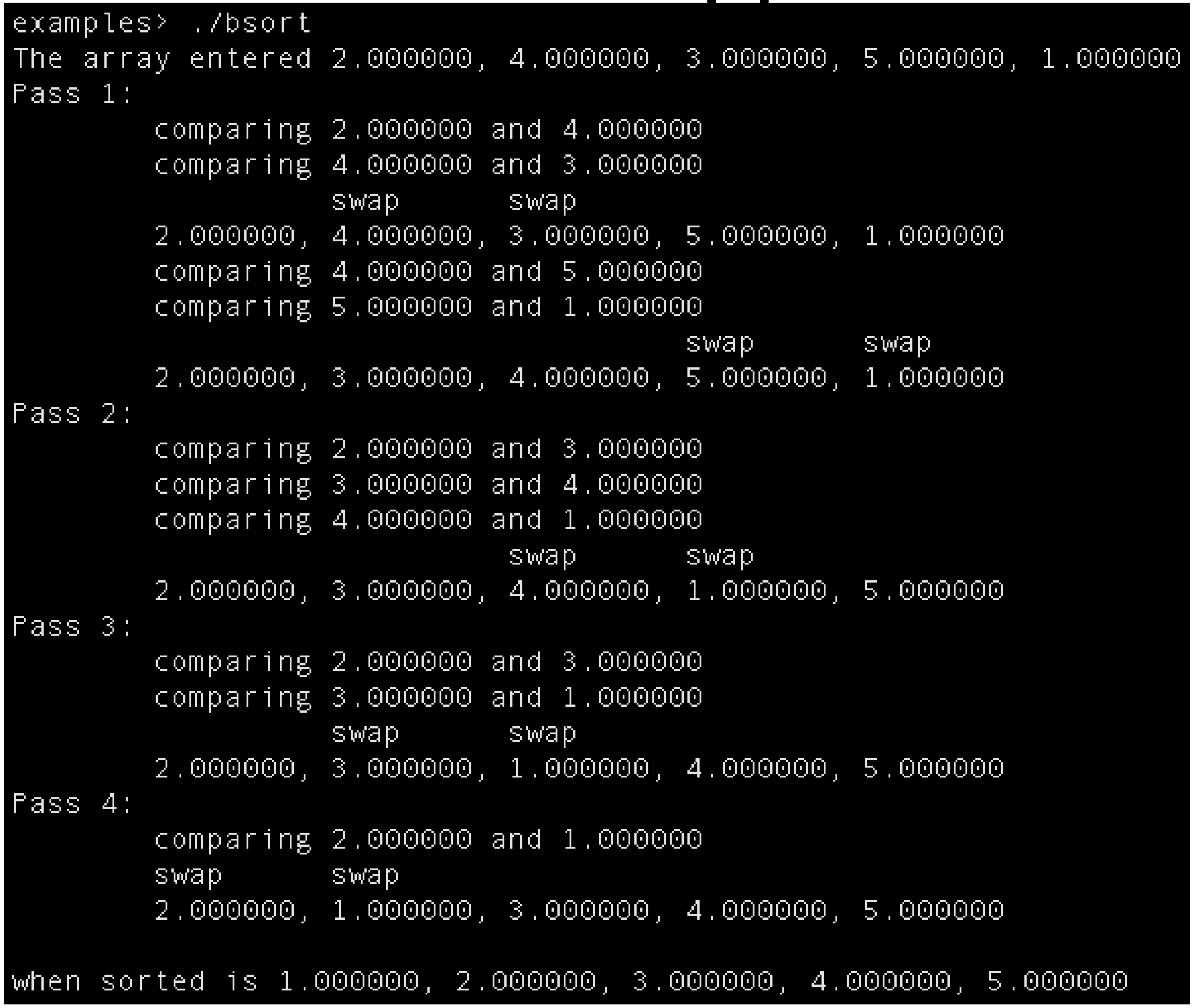


```
void bubble_sort(float a[], int length)
{
  for (int pass = 1; pass < length; pass++) {
    printf("Pass %d:\n", pass);
    for (int i = 0; i < (length - pass); i++) {
      printf("\tcomparing %f and %f\n", a[i], a[i+1]);
      if (a[i] > a[i + 1]) {
        print_sorting(a, length, i, i+1);
        swap(&(a[i]), &(a[i+1]));
      }
    }
  }
}
```



```
examples> ./bsort
The array entered 2.000000, 4.000000, 3.000000, 5.000000, 1.000000
Pass 1:
        comparing 2.000000 and 4.000000
        comparing 4.000000 and 3.000000
                  swap      swap
        2.000000, 4.000000, 3.000000, 5.000000, 1.000000
        comparing 4.000000 and 5.000000
        comparing 5.000000 and 1.000000
                                      swap      swap
        2.000000, 3.000000, 4.000000, 5.000000, 1.000000
Pass 2:
        comparing 2.000000 and 3.000000
        comparing 3.000000 and 4.000000
        comparing 4.000000 and 1.000000
                            swap      swap
        2.000000, 3.000000, 4.000000, 1.000000, 5.000000
Pass 3:
        comparing 2.000000 and 3.000000
        comparing 3.000000 and 1.000000
                  swap      swap
        2.000000, 3.000000, 1.000000, 4.000000, 5.000000
Pass 4:
        comparing 2.000000 and 1.000000
        swap      swap
        2.000000, 1.000000, 3.000000, 4.000000, 5.000000

when sorted is 1.000000, 2.000000, 3.000000, 4.000000, 5.000000
```

Bubble Sort Application



```
examples> ./bsort
The array entered 2.000000, 4.000000, 3.000000, 5.000000, 1.000000
Pass 1:
        comparing 2.000000 and 4.000000
        comparing 4.000000 and 3.000000
                  swap      swap
        2.000000, 4.000000, 3.000000, 5.000000, 1.000000
        comparing 4.000000 and 5.000000
        comparing 5.000000 and 1.000000
                                      swap      swap
        2.000000, 3.000000, 4.000000, 5.000000, 1.000000
Pass 2:
        comparing 2.000000 and 3.000000
        comparing 3.000000 and 4.000000
        comparing 4.000000 and 1.000000
                            swap      swap
        2.000000, 3.000000, 4.000000, 1.000000, 5.000000
Pass 3:
        comparing 2.000000 and 3.000000
        comparing 3.000000 and 1.000000
                  swap      swap
        2.000000, 3.000000, 1.000000, 4.000000, 5.000000
Pass 4:
        comparing 2.000000 and 1.000000
        swap      swap
        2.000000, 1.000000, 3.000000, 4.000000, 5.000000

when sorted is 1.000000, 2.000000, 3.000000, 4.000000, 5.000000
```

Bubble Sort is O(n^2)

- For each pass through the array
  - One needs to pass through most of the array.
- N passes through the array for n comparisons → O(n^2)

Median

Median

- The median is the number where half of the data values are greater than the number and half are lower.
  - Given a sorted array, the median is just the value in the middle of the array.
  - If the array is even, it is the average of the two middle values

Search

Search

- Finding an item in an array is O(n)
  - When the item isn't found, you need to look at all items in the array. (Worst case)
- Finding an item in a sorted array is O(log(n))
  - Look at the middle element
    - If it is higher the search item look in the first half
    - If it is lower, look in the second half

Binary Search

- binary_search(item, array, low, high)
  - If (item < array[low] or item > array[high])
    - Return not_found
  - mid = (high – low) div 2
  - If (item = array[mid])
    - Return mid
  - If (item < array[mid])
    - Return binary_search(item, array, low, mid)
  - If (item > array[mid])
    - Return binary_search(item, array, mid, high)

Does it work?

- The item is between array[low] and array[high] because array is sorted.
- If the item is between is the middle item it will be returned.
- If not the half it is in is searched
- All elements searched
  - Initially low = first element; high = last element
  - low <= mid < high

How long does it take

- At each step half of the array is ruled out.
  - I.e., Let n be the length of the array. The first search is the entire array $s_0(n)$. The second is half the array $s_1(n/2)$.
  - In general $s_i(n/2^i)$
- In the worst case, $n/2^i = 1$
  - I.e. it is the last place you look.
- In the worst case, $n = 2^i$
- $\text{Log}_2(n) = i$
- Binary search is O(log(n))