We’re gonna look at 4 algorithms today. One you’ve already seen, one that is *crazy*, and two that are hard but YOU CAN DO IT.
Some more details about Runtime Complexity, then work through MergeSort, then yell Insertion Sort and Selection Sort
Runtime Complexity

- Algorithms take time to run
- Clock time varies
  - Can vary based on input size / status (n = 1, 10, 100, 1000, 10000, 1000000, 1000000000000000)
  - Can vary based on number and kind of steps [swap is 3 assignments? 3*O(1) = O(1) ]
- Typically talk about runtime with “Big Thing“ notation
  - Big O - worst base - perfectly unsorted input
  - Big Omega - best case - perfectly sorted input
  - Big Theta - average case - fancy math that you’ll care about a year from now
    - Contrast with mergesort, which you will care about immediately next term.
# The Time Complexity Power Scale

<table>
<thead>
<tr>
<th>O(1)</th>
<th>Constant - simple assignment, condition checking, most math</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(log n)</td>
<td>Logarithmic - recursive calls on two halves</td>
</tr>
<tr>
<td>O(n)</td>
<td>Linear - for loops, while loops</td>
</tr>
<tr>
<td>O(n log n)</td>
<td>“n log n” - loops ‘inside’ of recursive call</td>
</tr>
<tr>
<td>O(n^2)</td>
<td>Quadratic - nested loops</td>
</tr>
<tr>
<td>O(n^3)</td>
<td>Cubic - triple nested loops</td>
</tr>
<tr>
<td>n^O(1)</td>
<td>Polynomial - P vs NP</td>
</tr>
<tr>
<td>2^O(n)</td>
<td>Exponential - over 9000</td>
</tr>
</tbody>
</table>
O(n^2) algorithm example: bubble_sort

```c
void bubble_sort(struct node *head, int size) {
    ...
    int iteration, i
    for(iteration=1; iteration<size; iteration++) {
        for(i=0; i<size-iteration; i++) {
            if(current->val > current->next->val) {
                //swap values
            }
            //move current to next node
        }
        //move current to next node
        current = head;
    }
}
```
Merge Sort

https://www.youtube.com/watch?v=7LN9z140U90 - whole video

1] Divide array into two halves, make sure you can (if left < right)
2] Sort left half with a recursive call (left to mid)
3] Sort right half with a recursive call (mid to right)
4] merge the two sorted halves
Merge - Overwriting an array $A$ with two temp stacks

1. Make two stacks, a lower_half and an upper_half

2. Loop: step to end of each stack in parallel
   a. If lower stack’s number is smaller than upper stack’s, pop lower stack’s number onto $A[ ]$
   b. Otherwise, pop the upper stack’s num onto $A[ ]$
   c. Check if we’re at the end of one of our stacks
      i. If at end of lower, pop the rest of upper
      ii. If at the end of upper, pop the rest of lower
      iii. If neither, restart loop

3. Done!
MergeSort complexity analysis

https://youtu.be/0nlPxaC2ITw?t=269 - 4:29 until 10:39

Simple explanation:

- Mergesort uses a left/right half recursive call, which is $O(\log(n))$
- Each recursive call must call merge, which is $O(n)$
- Recursive calls ‘contain’ for loop, so:
  - $\text{mergesort(left)}$ is $O(n \log n)$
  - $\text{mergesort(right)}$ is $O(n \log n)$
  - $\text{merge()}$ is $O(n)$
  - $\text{mergesort(n)} = O(n \log n) + O(n \log n) + O(n) = O(n \log n)$
void insertion_sort(int *nums, int size) {
    int i, j, temp;

    //What does this loop do?
    for(i=0; i<size; i++) {
        temp=nums[i];
        //What does this loop do?
        for(j=i; j>0 && nums[j-1]>temp; j--)
            nums[j]=nums[j-1];
        //What does this statement do?
        nums[j]=temp;
    }
}
Selection Sort

void selection_sort(int *nums, int size) {
    int i, j, temp, min;
    //What does this loop do?
    for(i=0; i<size-1; i++) {
        min=i;
        //This one?
        for(j=i; j<size; j++)
            if(nums[j]<nums[min])
                min=j;
        //What elements are being swapped?
        temp=nums[i];
        nums[i]=nums[min];
        nums[min]=temp;
    }
}
```c
int binarySearch(const int list[], int length, int item) {
    int first = 0, last = length - 1, mid;
    int found = 0;
    while (first <= last && found == 0) {
        mid = (first + last) / 2;
        if (list[mid] == item)
            found = 1;
        else if (list[mid] > item)
            last = mid - 1;
        else
            first = mid + 1;
    }
    return found==1?mid:-1; // if found return mid, else return -1 (not in list[])
} //end binarySearch
```