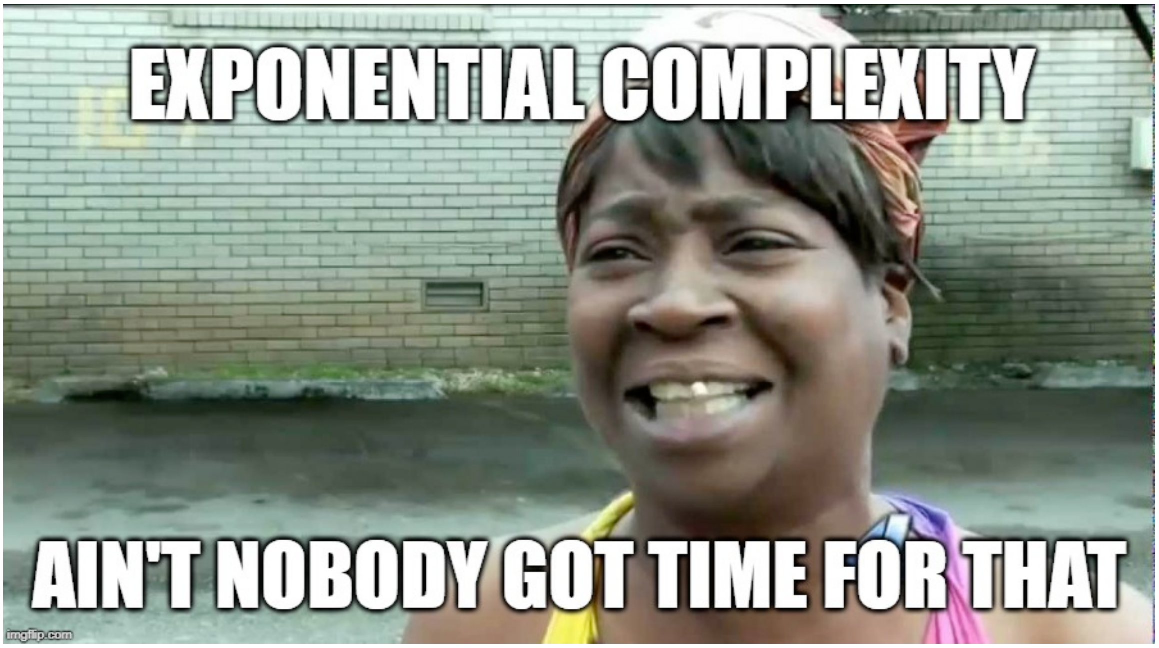
Week 7



Announcements

- Good job on midterm 2!
- Project 3
    - Tuesday 5/18
- HW 4
    - Tuesday, 5/25

LA Program

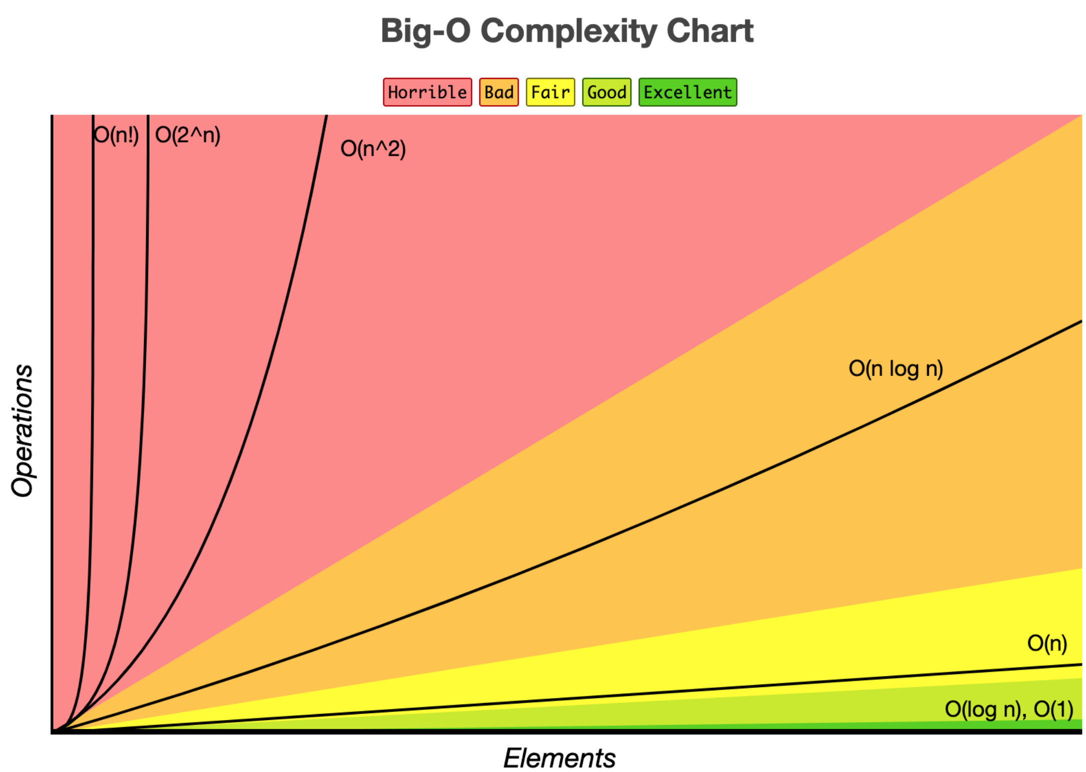
- https://ceils.ucla.edu/learningassistants/for-prospective-learning-assistants/

Questions?

- anything?
- minimax, undo function

Big-O

- helps us reason about the complexity of our code
  - how many operations does it execute
  - not exact
- e.g if I want to sort N numbers
  - should I choose a $O(N^2)$ or a $O(N)$ algorithm?
- $O(N^2 + 5N + 28) = O(N^2)$
- $log_2(N) = log(N)$
- for small N, doesn't matter as much (but big n can be drastically different)
  - super useful cheat sheet: https://www.bigocheatsheet.com/



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Examples

```
// look at the inputs, write in terms of inputs

/****************************************/

// complexity: O(1)
void foo() {

  int x = 5;
  int y = x*x;
  cout << y << endl;

}

// complexity: O(1)
```

```
void bar(int a[], int n, int pos) {

  cout << a[pos] << " ";

  int r = 5 + 10;

}

/*****************************************/
// for loop

// complexity: O(n)
void baz(int a[], int n) {

  for(int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }

}

/*****************************************/
// nested for loop, 2 variables

// complexity: O(n*m)
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    cout << arr2[i][j] << " ";
  }
}

// O( c^2 + e)
void bumble(string csmajors[], int c, string eemajors[], int e) {

  for (int i = 0; i < c; i++)
    for (int j = 0; j < c; j++)
      cout << csmajors[i] << csmajors[j];

  for (int i = 0; i < e; i++)
    cout << eemajors[i]

}

/*****************************************/
// bit trickier

// complexity: O( n^2 )
for (int i = 0; i < n; i++) {

  for (int j = 0; j < i; j++) {
    cout << arr2[i][j] << " ";
  }

}
```

```
0 + 1 + ... + n-1 ~ n^2

/**************************************/
// a different function

// complexity: O( n * log(n) )
for (int i = 0; i < n; i++) {

  int k = i;
  while (k > 1) {
    // do something
    k /= 2;
  }

}

/**************************************/

// stl example
/*
  study the time complexity of different functions in STL containers
  for interviews you should at least know the cost of searching so you can
  choose the best data structure for your task.
*/
```

Space Complexity

```
// how much space you use as a function of your parameters

// O(1)
void foo(int a[], int n) {
  int i = 5;
  // ... other stuff
}

// O(n)
void bar(int a[], int n) {
  int* arr = new int[n]
  // ... other stuff
}

// loop vs recursion

// space complexity: O(1)
void sumNums(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += i;
}
```

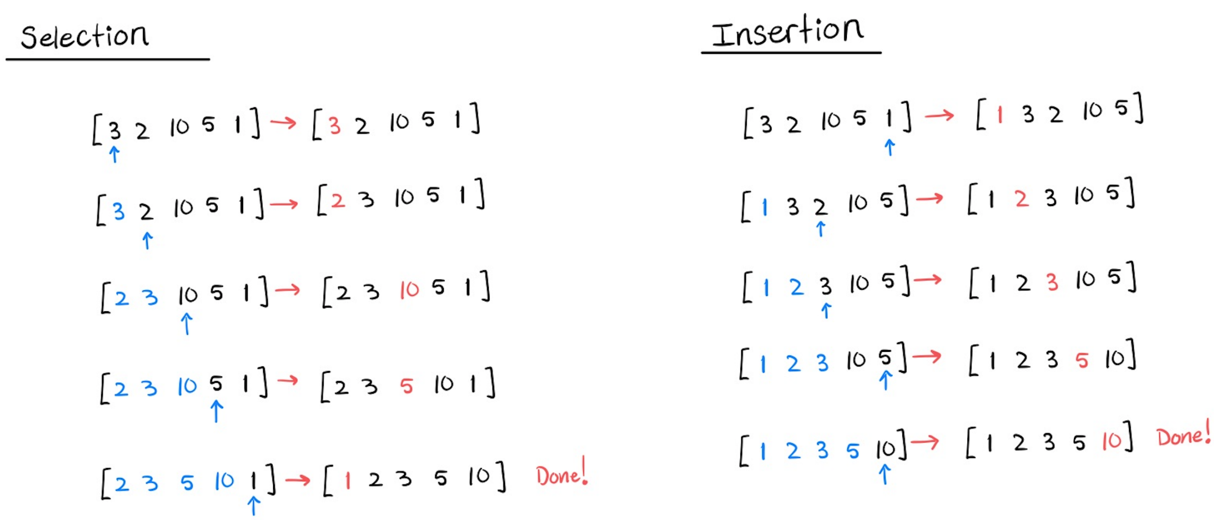
```
// space complexity: O(n)
int sumNums(int n) {
  if (n == 0 || n == 1)
    return n;

  return n + sumNums(n-1);
}

// now you can see how recursion can be less efficient in terms of space usage
```

(Bad) Sorting Algorithms

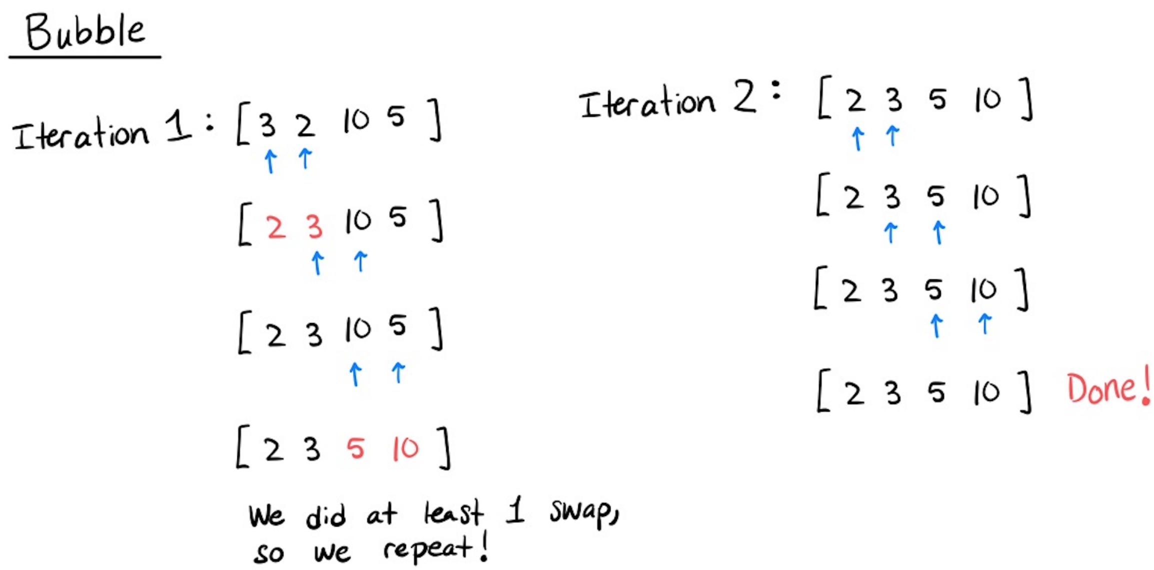
```
//** Selection **//
/*
  average - O(n^2)
  best case (already ordered) - O(n^2) still not efficient because it has to
                                        compare all unsorted elements
  worst case (reverse order)  - still O(n^2)

  stable - no, because we swap items
*/

//** Insertion **//
/*
  average - O(n^2)
  best case (already ordered) - more efficient because we don't have to do
                                as many comparisons O(n)
  worst case (reverse order) - O(n^2)

  stable - yes, relative order is maintained
*/


//** Bubble **//
/*
  average - O(n^2)
  best case (already ordered) - we have to run the algorithm fewer times, O(n)
  rworst case (reverse order) - O(n^2)

  stable? - no, because we're swapping elements
*/
```

- For selection and insertion sort, we're basically creating a sorted array at the front, one element at a time (represented by blue numbers)
    - Here I'm accumulating the smallest values, but in class you guys worked with moving the larger numbers to the back

- selection sort we pick the next number in our unsorted list (the black numbers) and put it in the right spot in the sorted array (the blue numbers).
- insertion sort, we pick the smallest of the unsorted (black) numbers and put them at the end of the sorted numbers (blue).



For bubble sort, we compare pairs of numbers. Below you can see that the 10 "bubbles" up to the end. After each iteration of bubble sort, our next largest number is pushed towards the end.



Shell

- bit better than bubble, but still not that good
  - between O(n) and O(n^2)

"Good" Sorts

- divide and conquer, usually O(n* log(n))
  - O(log(n)) levels, O(n) each level

Quick Sort

- this is illustrating the partition algorithm, not the entire quicksort algorithm
- worst case: O(n^2)

pivot first last

30 1 77 13 99 12 35

f l

30 1 77 13 99 12 35

f l

30 1 77 13 99 12 35 swap!

f l

30 1 12 13 99 77 35

f l

30 1 12 13 99 77 35

f l

30 1 12 13 99 77 35

f l

30 1 12 13 99 77 35

l f

30 1 12 13 99 77 35 l is before f

l f

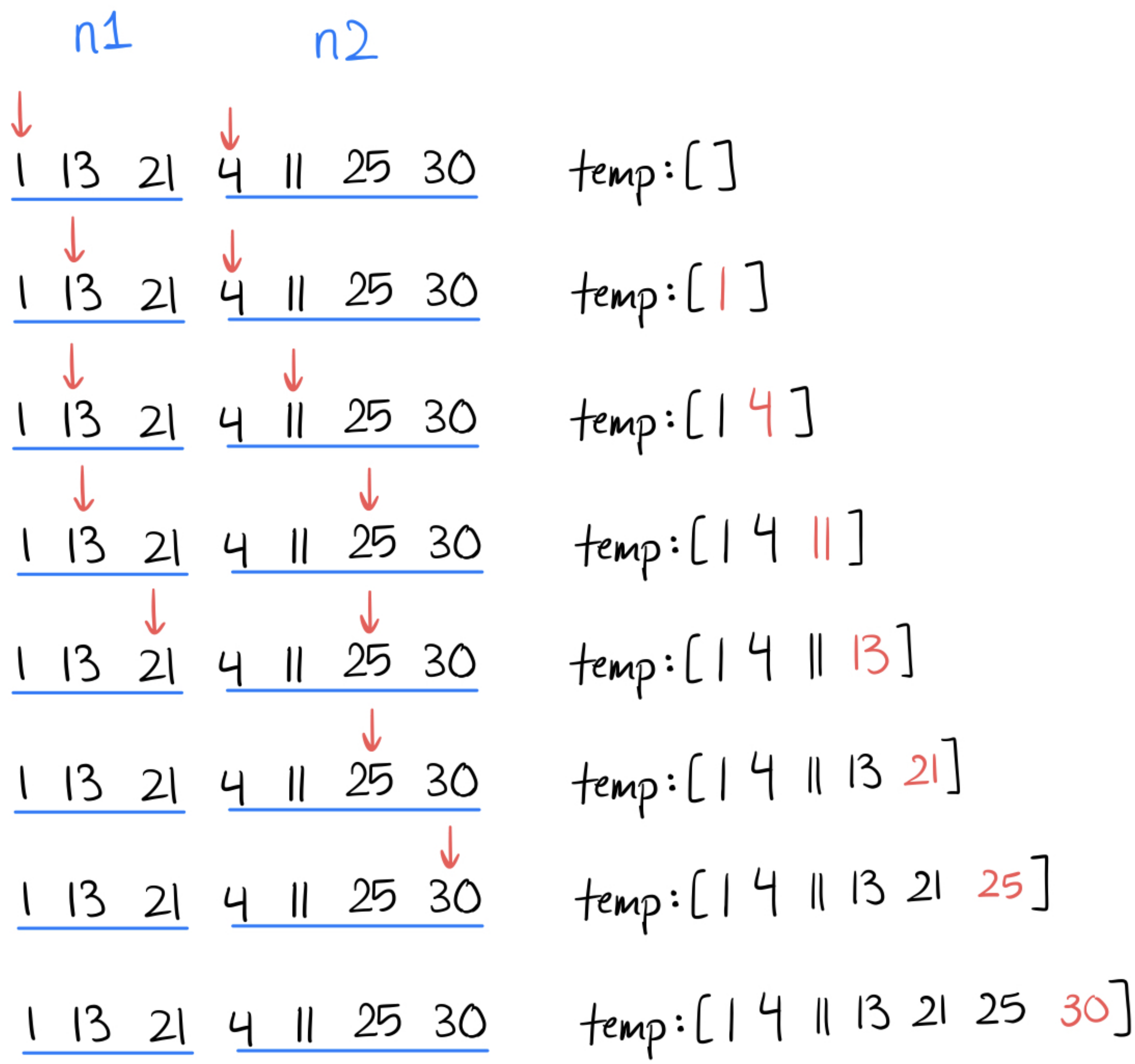
13 1 12 30 99 77 35 now recurse!

```
// quicksort

/*
select p, usually first element
move p to the middle (all elements to the left are less than or equal to p)
    to the right, everything is greater
    now recursively divide left and right
*/

void quicksort(int a[], int start, int end) {
  if (end - start >= 1) {
    int pivotIndex;

    // dificult part
    pivotIndex = Partition(a, start, end);

    // recursion, divide and conquer
    quicksort(a, start, pivotIndex-1);
    quicksort(a, pivotIndex, end);
  }
}

// complexity? O( )
int partition(int a[], int low, int high) {
  int pi = low;
  int pivot = a[low]; // chooses first element of array

  do {

    // look for a value larger than pivot (belongs on the right side)
    while ( low <= high && a[low] <= pivot)
      low++;

    // look for a value smaller than pivot (belongs on the left side)
    while ( a[high] > pivot )
      high--;

    // swap the two elements we've found that are on the wrong sides
    if (low < high)
      swap(a[low], a[high]);

  } while (low < high)

  // at the end, swap the pivot element with the element pointed to by high
  swap(a[pi], a[high]);

  pi = high;

  return pi;

}
```

```
// worst case? if perfectly sorted, O(n^2)
// stable? no (swapping)
```

Merge Sort

- again, this is just the merge algorithm and not the entire mergesort algorithm




```
// mergesort

/*
  pseudocode
    if arr size == 1, return (base case, already sorted)
```

```
    split array into two equal parts
      reursively call mergesort on 1st half
      reursively call mergesort on 2nd half

    merge the two sorted halves
*/
```

```
void merge(int data[], int n1, int n2, int temp[]) {
  int i1 = 0, i2 = 0;
  int k = 0;
  int* A1 = data, *A2 = data+n1;

  while (i1 < n1 || i2 < n2) {
    // exhausted one arrray, just copy the rest over
    if (i1 == n1)
      temp[k++] = A2[i2++];
    else if (i2 == n2)
      temp[k++] = A1[i1++];

    // choose the smaller element
    else if (data[i1] <= A2)
      temp[k++] = A1[i1++];
    else
      temp[k++] = A2[i2++];
  }

  // replace data with temp
  for (int i = 0; i < n1+n2; i++)
    data[i] = temp[i]

}

// need O(n) storage
// no worst case, but slow b/c extra memory
// stable? yes
```