CS 161 - Section 2

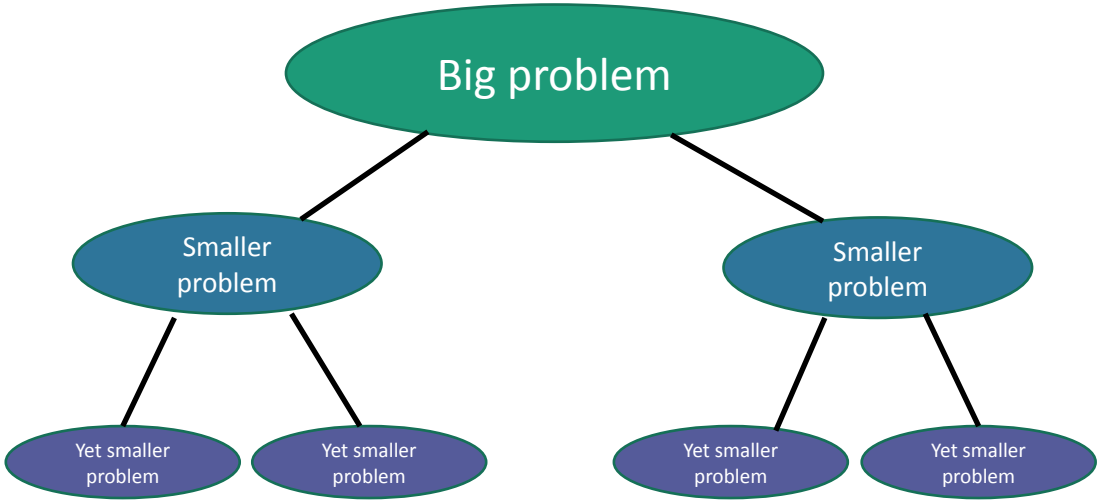
Section 2 agenda

- Recap lectures:
  - Select
  - Recurrence Relations
  - Master Theorem
  - Quick Sort
- Space complexity
- Handout

Sorting and Selecting

Main idea: Divide and Conquer

- **Merge sort** – divide and conquer algorithm for sorting an array
- **SELECT** – divide and conquer algorithm for finding the *k*th smallest element of an array



SELECT

- **`getPivot`**`(A)` returns some pivot for us.
- **`Partition`**`(A,p)` splits up A into L, A[p], R.

- **`Select`**(A,k):
  - **If** len(A) <= 50:
    - A = **`MergeSort`**(A)
    - **Return** A[k-1]
  - p = **`getPivot`**(A)
  - L, pivotVal, R = **`Partition`**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **`Select`**(L, k)
  - **Else if** len(L) < k-1:
    - return **`Select`**(R, k – len(L) – 1)

Base Case: If the len(A) = O(1), then any sorting algorithm runs in time O(1).

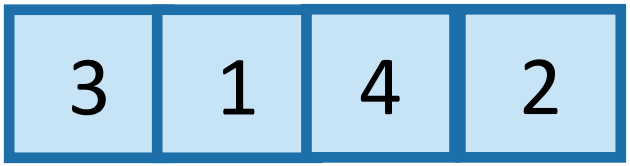
Case 1: We got lucky and found exactly the k'th smallest value!

Case 2: The k'th smallest value is in the first part of the list

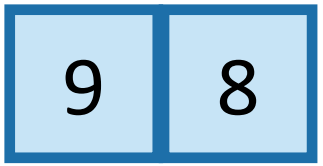
Case 3: The k'th smallest value is in the second part of the list

Partitioning

Say we want to find SELECT(A, k)



L = array with things smaller than A[pivot]



R = array with things larger than A[pivot]

- If k = 5 = len(L) + 1:
  - We should return A[pivot]
- If k < 5:
  - We should return SELECT(L, k)
- If k > 5:
  - We should return SELECT(R, k – 5)

Partitioning like this takes time O(n) since we don't care about sorting each half.

Choosing the pivot

- CHOOSEPIVOT(A):
  - Split A into m = $\lceil \frac{n}{5} \rceil$ groups, of size <=5 each.
  - **For** i=1, .., m:
    - Find the median within the i'th group, call it p_i
  - p = SELECT([$p_1, p_2, p_3, \ldots, p_m$] , m/2)
  - **return** p

This takes time O(1), for each group, since each group has size 5. So that's O(m)=O(n) total in the for loop.

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

Pivot is SELECT(| 8 | 4 | 5 | 6 | 12 | , 3) = 6:

1	8	9	3	15	5	9	1	3	4	12	2	1	5	20	15	13	2	4	6	12	1	15	22	3

PARTITION around that 6:

1	3	5	1	3	4	2	1	2	4	1	3	5	6	8	9	15	9	12	20	15	13	12	15	22

This part is L

This part is R: it's almost the same size as L.

SELECT

- **`getPivot`**`(A)` returns some pivot for us.
  - How?? Median of sub-medians!
- **`Partition`**`(A,p)` splits up A into L, A[p], R.

- **`Select`**(A,k):
  - **If** len(A) <= 50:
    - A = **`MergeSort`**(A)
    - **Return** A[k-1]
  - p = **`getPivot`**(A)
  - L, pivotVal, R = **`Partition`**(A,p)
  - **if** len(L) == k-1:
    - return pivotVal
  - **Else if** len(L) > k-1:
    - return **`Select`**(L, k)
  - **Else if** len(L) < k-1:
    - return **`Select`**(R, k – len(L) – 1)

Running time:

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + \Theta(n)$$

Recurrence Relations

- Divide and conquer algorithms are often recursive in nature – need to know how to solve recurrence relations for runtime analysis

- Two methods:
  - Master Method
  - Substitution Method

The Master Method

Remember that we can (mostly) ignore floors and ceilings in this class: $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$ work too.

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. Then

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Three parameters:
a : number of subproblems
b : factor by which input size shrinks
d : need to do n^d work to create all the subproblems and combine their solutions.

Requires all subproblems to be the same size!

The Substitution Method

1. Guess what the answer is.
   - Try a few levels of recursion and see if you spot a pattern
2. Formally prove that that's what the answer is.
   - Often using induction

Space Complexity

Warm-up

- How many **bits** do you need to store a number between 1 and n?
  - Answer is log(n), since k bits can represent all numbers between 1 and 2^k
- How many **digits** do you need to represent that same number?
  - $\log_{10}(n)$
- What about **base-r digits**?
  - $\log_r(n)$

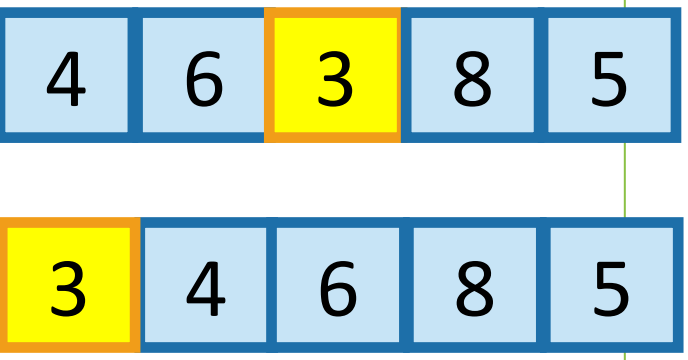
Keep this in the back of your mind for Week 3!

Space complexity of an algorithm

- Definition: the **space complexity** of an algorithm is how much memory the algorithm needs to run, excluding the input and output.

- Expressed as a function of input size
  - Could vary based on language, compiler, etc. Big-O notation!

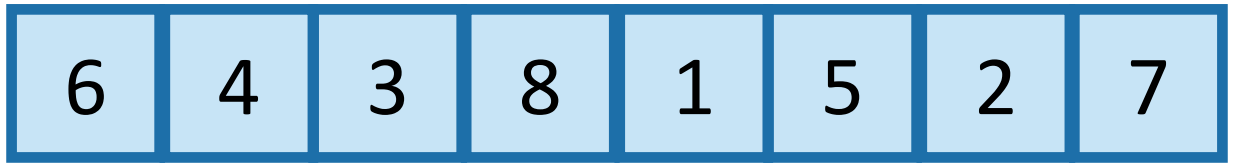
Example: Insertion Sort

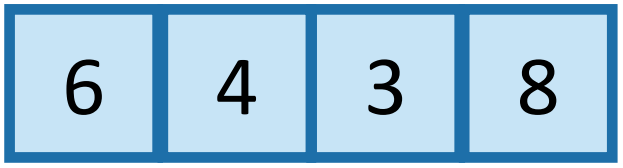



```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

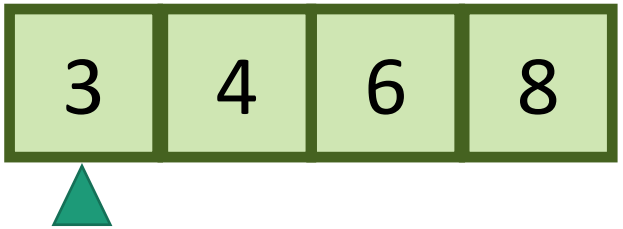
- Input: array of size n
- All operations are done in-place □ no extra space needed
- Space complexity = O(1)

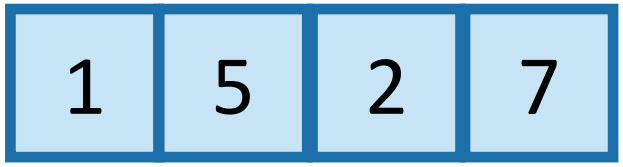
Example: Merge sort



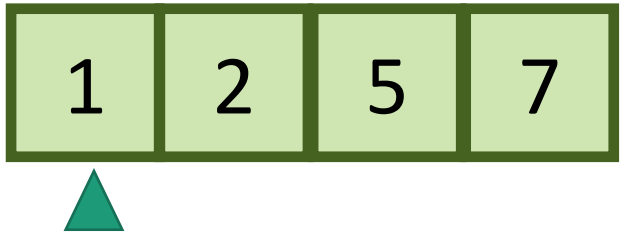


Recursive magic!

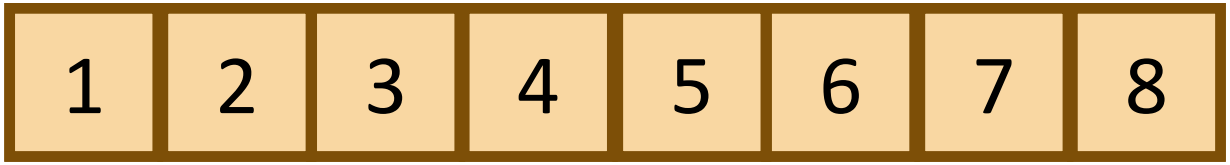




Recursive magic!



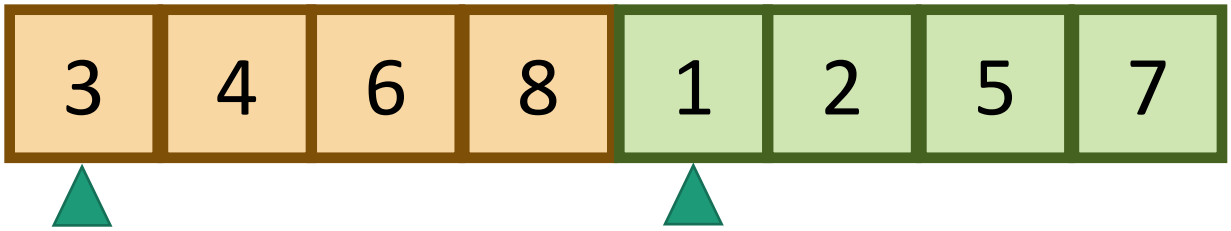
MERGE!



Example: Merge sort

- Merging two arrays of size k/2 into a new array of size k requires extra space of size k
- The top level of merge sort needs space n, so merge sort has space complexity O(n)
  - Merge sort has log(n) levels of merges, why is it not n log(n)?
- **Can we do better?**

In-place merging



- If the left element is smaller, move the left pointer to the right.
- If the right element is smaller, move it to the position of the left element and shift everything in between to the right. Then move both pointers to the right.
- Now requires no extra space space complexity is O(1)!

In-place merging

- What happened to time complexity?
  - "Shift everything to the right" is O(n), in the worst case we need to do it O(n) times
- This merge takes time $O(n^2)$!

- Often there is a trade-off between time and space complexity.
  - In what situations is having a small space complexity more important?

Quick Sort

QuickSort

- QuickSort(A):
  - **If** len(A) <= 1:
    - **return**
  - Pick pivot x with **pivot**.
  - PARTITION the rest of A into:
    - L (less than x) and
    - R (greater than x)
  - Rearrange A as [L, x, R]
  - QuickSort(L)
  - QuickSort(R)

Running time:

$$T(n) = T(|L|) + T(|R|) + \Theta(n)$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

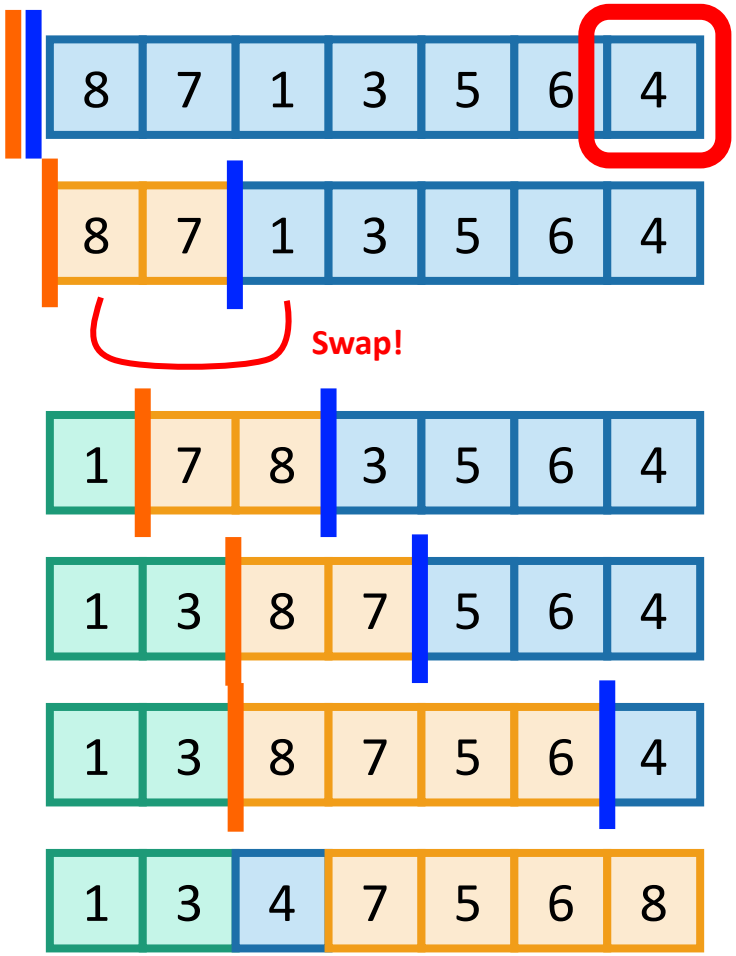
$$\boxed{T(n) = O(n\,log(n))}$$

In-Place [O(1) memory!] Quick Sort

- Recall the Naïve memory complexity of Quick Sort is O(n logn)
  - Why? Think about storing an ordering of n elements for log(n) levels
- We can improve it to O(n)
  - Why? Can use a single array to represent the ordering and update at each level
- Can we do even better?
  - Let these happy Hungarians show you the answer!

https://www.youtube.com/watch?v=ywWBy6J5gz8&ab_channel=AlgoRythmics

A better way to do Partition



Initialize and

Step forward.

When sees something smaller than the pivot, **swap** the things ahead of the bars and increment both bars.

Repeat till the end, then put the pivot in the right place.

Quick Sort vs Merge Sort

	QuickSort (random pivot)	MergeSort (deterministic)
Running time	• Worst-case: $O(n^2)$ • Expected: O(n log(n))	• Worst-case: O(n log(n))
In-Place? (With O(log(n)) extra memory)	Yes, can be implemented in-place (relatively) easily	Not as easily since you'd have to sacrifice stability and runtime, but it can be done
Stable?	No	Yes

stable sorting algorithms sort identical elements in the same order as they appear in the input

Quick Sort vs Merge Sort

Which one would you use for a small array?

Given the small size it mostly does not matter. You could still argue for Quick sort as the *expected* runtime is O(n logn) but we can get away with faster than that, and even if we miss, the worst case is O(n^2). But at this scale the difference will be in the order of *ms*

Which one would you use for an array with millions of elements?

Because for large *n* the Law of Large Numbers kicks in, we can reasonably expect both algorithms to run in O(n logn). It then becomes a priority choice between stability and memory-space

Which one would you use for an array of unknown size?

This might be a little of a personal choice, but *usually* for unknown inputs (assuming you don't even know the expected range of sizes) we value the predictability and consistency of a deterministic algorithm, so Merge sort would be preferred. Randomized algorithms can be quite challenging to debug.

Thank You!