Main topics are divide-and-conquer including example/quicksort and example/selection.

Quicksort

- Today, we’re going to finish up the divide-and-conquer part of the course by looking at quicksort in a bit more detail and discussing the related problem of median selection.
- Here’s the pseudocode for quicksort. Partition(A[1 .. n], p) takes the index of the pivot element as its second parameter and returns the new index of the pivot element after partitioning.

```
QuickSort(A[1 .. n]):
   if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1 .. r - 1])  \(\text{}}\) (Recurse!)
      QuickSort(A[r + 1 .. n])  \(\text{}}\) (Recurse!)
```

Analysis

- You can argue for correctness of Partition and QuickSort using induction, but how fast is it?
- Like in MergeSort, we need a recurrence, but now the size of the recursive calls depends upon the rank of the pivot element.
- The rank of an element in A[1 .. n] is its index after you sort the array. So the smallest element has rank 1 and the largest element as rank n.
- The median element has rank ceil{n / 2}.
- Partition moves all smaller elements to the left of the pivot and larger elements to the right. So the value r it returns is its rank.
- If we knew r would take on a certain value, we could write a worst-case running time recurrence of T(n) = T(r - 1) + T(n - r) + O(n), because Partition is just a for loop over n - 1 elements.
- The best thing would be if the pivot was the median element, meaning r = ceil{n / 2}. Then
  - T(n) = T(ceil(n / 2) - 1) + T(floor(n / 2)) + O(n) ≤ 2T(n / 2) + O(n).
- And we already know the solution to that recurrence is O(n log n).
- As we’ll see in a few minutes, picking the median element is actually kind of tricky.
- So what usually happens is programmers will do something simple like pick the first or last element as the pivot. But now you have no control over the rank of the pivot. The worst-
case running time follows the recurrence

- \( T(n) = \max_{1 \leq r \leq n} (T(r - 1) + T(n - r) + O(n)). \)
- In particular, we’re worst off when the two subproblems are completely unbalanced, at every level of the recursion tree, i.e. \( r = 1 \) or \( r = n \). Now, \( T(n) = T(n - 1) + T(1) + O(n) = T(n - 1) + O(n). \) [Use the right recursion tree.]

This recurrence solves to \( T(n) = O(n^2) \). That’s not great.

There are other ways to pick the pivot that tend to work better. A popular choice is to pick the median of the three elements such as one each from the beginning, middle, and end of the array. But in the worst case, you still get a really unbalanced recurrence that solves to \( O(n^2) \).

So it turns out quicksort is one of those algorithms that works really well in practice, but it probably runs in Theta\((n^2)\) time in the worst-case, depending upon your implementation.

**Selection**

- But let’s say we really do want to find that median element. We might do so not just for the sake of quicksort, but also because medians are a fundamental statistic you might need when studying different kinds of data.
- It turns out the median is no easier to find than any other element. In fact, when we start using recursion, we’ll wish we had a way to find other elements.
- So let’s solve the more general problem of element selection. For this problem, we’re given an array \( A[1 .. n] \) and an integer \( k \) where \( 1 \leq k \leq n \).
- We want to find the element of rank \( k \) in \( A \).
- We could solve this problem by sorting the array and then picking the element now in position \( k \), but that would take \( O(n \log n) \) time.
- We can do better, though, by taking some hints from the sorting algorithms we’ve looked at. Tony Hoare described an algorithm we’ll call quickselect or one-armed quicksort on the same page where he first published quicksort.
- Quickselect is kind of like a binary search. We choose an arbitrary pivot element like in quicksort. Then we call Partition to learn the rank of our pivot. Since we partitioned the array around the pivot, we can now recursively search the half of the array containing the
element of rank k.

\[
\text{QuickSelect}(A[1..n], k): \\
\quad \text{if } n = 1 \\
\quad \quad \text{return } A[1] \\
\quad \text{else} \\
\quad \quad \text{Choose a pivot element } A[p] \\
\quad \quad r \leftarrow \text{Partition}(A[1..n], p) \\
\quad \quad \text{if } k < r \\
\quad \quad \quad \text{return } \text{QuickSelect}(A[1..r - 1], k) \\
\quad \quad \text{else if } k > r \\
\quad \quad \quad \text{return } \text{QuickSelect}(A[r + 1..n], k - r) \\
\quad \quad \text{else} \\
\quad \quad \quad \text{return } A[r]
\]

- Notice how we have to change the second parameter in the second recursive call to take into account that \(A[r + 1 .. n]\) is missing the smallest \(r\) elements of \(A[1 .. n]\).
- Like quicksort, the correctness of the algorithm does not depend on the choice of pivot. Unfortunately, the running time does!
- The worst-case running time follows the recurrence
  - \(T(n) = \max_{1 \leq r \leq n} \max \{T(r - 1), T(n - r)\} + O(n)\).
  - And like before, the worst thing that can happen is that \(r = 1\) or \(r = n\), meaning \(T(n) = T(n - 1) + O(n)\). Again, this solves to \(T(n) = O(n^2)\)!
- But! If we somehow choose a pivot closer to the middle so that we recurse on alpha \(n\) elements for some constant \(a < 1\), then \(T(n) \leq T(a \cdot n) + O(n)\). The level-sums of the recursion tree decrease exponentially, so \(T(n) = O(n)\).
- In other words, if we can find a pivot element that’s even close to the median, then we can find the exact median in only \(O(n)\) time. We need an Approximate Median Fairy.
- In the early 1970s, Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan described how to implement the Approximate Median Fairy by computing the median of a carefully selected and much smaller subset of the input array. The Approximate Median Fairy is really the Recursion Fairy!
- To reiterate, they use recursion to pick a good pivot so they can use recursion to find the rank \(k\) element. Two different goals, but both solved by running the algorithm recursively.
- What we’ll do for the pivot selection is to partition the array into \(\text{ceil}(n/5)\) /blocks. We’ll compute the medians of each of those blocks by “brute force”, stick the medians in their own array \(M\), and then use recursion to find the median of \(M\). This “median of medians” will be our pivot.
**Analysis**

- The algorithm is correct, because it’s just a fancy way of picking a pivot for Quickselect, but what is the running time?
- The key insight is that we actually are picking a good pivot as described above.
- To see why, imagine we draw the input array as a 5 x ceil(n/5) grid. Each column represents five consecutive elements from the array.
- However, for illustration imagine we sort each column from top down. And then we sort the columns themselves by their middle element. **AGAIN, THE ALGORITHM ITSELF DOES NOT ACTUALLY SORT ANYTHING**

So here’s the median of these medians, right in the middle.

- Suppose the element we’re looking for is larger than the median of medians. In the recursive call to MomSelect, we’ll ignore all elements smaller than the mom.
- All those ceil(ceil(n / 5) / 2) ~ n / 10 medians to the left are smaller. And there are 3 elements per column that are at least as small as those. So the mom is larger than about 3n / 10 elements.
The recursive call will therefore involve at most \(7n/10\) elements. If the rank \(k\) element is smaller than the mom, a symmetric argument applies.

So we have a good pivot for a Quickselect, but now we’re doing two recursive calls instead of one. The other call uses about \(n/5\) elements.

So \(T(n) \leq T(n/5) + T(7n/10) + O(n)\).

Here’s the recursion tree. [On the left.]

- The root gets \(n\). It has two children of value \(n/5\) and \(7n/10\).
- If we write out a couple levels, we see level \(i\) sums to \((9/10)^i\) \(n\). It’s a decreasing geometric series, so \(T(n) = O(n)\). Hurray!
- But why \(5\)? Well, even numbers cause other complications, and \(5\) is the smallest odd block size that gives us a decreasing geometric series in the running time.
- For example, if we used blocks of size \(3\), then we would discard about \((n/3)/2 = n/3\) locations in the second recursive call, so it would operate on at most \(2n/3\) elements. But the first call would use \(n/3\) elements, giving a recurrence of \(T(n) \leq T(n/3) + T(2n/3) + n\).
- And as we saw last Wednesday, the solution to this recurrence is \(O(n \log n)\). We may as well just sort the array in this case!
- Now, having said all that, the constants in the \(O(n)\) for MomSelect are pretty big. And in practice, quickselect is very fast if you pick a reasonable pivot. So you probably wouldn’t implement MomSelect for use in practice.
- But the real best thing to do for both quicksort and quickselect is to pick the pivot element uniformly at random. Then the expected running time of quickselect, even for the worst possible input array, would be about a quarter of the worst case running time for MomSelect. A random pivot for quicksort leads to a worst-case \(O(n \log n)\) expected running time.