CS 6363.005.19S Lecture 6—January 31, 2019

Main topics are #divide-and-conquer with #example/fast_Fourier_transforms and #example/selection.

Prelude

- We have a TA! Jiashuai Lu will hold office hours once a week, grade homework, and help me with other things in the course. He's passed the QE, so he does know what he's doing.
- His office hours are Wednesday 3-5pm in 2.104A1.
- Homework 1 is due Tuesday, February 5th. I have office hours this afternoon from 2-3 and Monday from 2-3 if you'd like help with the homework or anything else related to the class.

Finishing up FFTs

- Last time, we were discussing the fast Fourier transform.
- We're a polynomial p of degree at most n - 1 represented as an array of coefficients P[0 .. n-1] where P[j] is the coefficient a_j of x^j.
- Our goal is to find a sample representation of p by evaluating p at n different values of x. In doing so, we can multiply polynomials of degree n-1 faster than O(n^2) time, because multiplying two sample representations takes only linear time.
- It turns out we can compute the sample representation using divide-and-conquer if we choose the complex nth roots of unity for our sample positions. They all have the form w_n^k = e^{(2pi i / n) k} = cos (2pi / n * k) + i sin (2pi / n * k).
- The *discrete Fourier transform* is set of values obtained by evaluating p at the complex nth roots of unity. It can be represented as an array P^*[0 .. n-1] where

$$P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k] \cdot \omega_n^{jk}$$

- Our divide-and-conquer approach was to create polynomials consisting of only the even or odd coefficients of p, recursively evaluate the *squares* of the nth roots of unity in these new polynomials, and then use the resulting values to evaluate p at all n roots. Here is the pseudocode.

```
RADIX2FFT(P[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U[j] ← P[2j]
      V[j] ← P[2j+1]

  U* ← RADIX2FFT(U[0..n/2−1])
  V* ← RADIX2FFT(V[0..n/2−1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2−1
      P*[j]       ← U*[j] + ω·V*[j]
      P*[j+n/2] ← U*[j] − ω·V*[j]
      ω ← ω·ω_n

  return P*[0..n−1]
```

- Now, our original motivation for converting to sample representation was so we could multiply polynomials quickly, so it would be good to get back to coefficient representation.
- Recall the Vandermonde matrix. Since we're using the complex nth roots of unity, it looks like

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

- If a-> <a_0, a_1, …, a_{n-1}> is the set of coefficients and y-> = <y_0, y_1, …, y_{n-1}> is the discrete Fourier transform, then Va-> = y->.
- To go from samples y-> to coefficients a->, we need compute V^{-1} y-> ,but one can show the amazing fact V^{-1} = bar{V} / n. In other words, the (j, k) entry of V^{-1} is bar{w_n^k} / n = w_n^{-jk} / n.
- So inverting the discrete Fourier transform is the same as taking the discrete Fourier transform, except our roots of unity go around the unit circle in the opposite direction, and we need to divide our products by n.
- Here's what it looks like if we apply those changes directly to Radix2FFT.

```
INVERSEFFT(P*[0..n-1]):
  P[0..n-1] ← FFT(P*)
  for j ← 0 to n-1
      P*[j] ← conj(P*[j])/n
  return P[0..n-1]
```

```
INVERSERADIX2FFT(P*[0..n-1]):
  if n = 1
      return P
  for j ← 0 to n/2-1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]
  U ← INVERSERADIX2FFT(U*[0..n/2-1])
  V ← INVERSERADIX2FFT(V*[0..n/2-1])
  ω_n ← cos(2π/n) - i sin(2π/n)
  ω ← 1
  for j ← 0 to n/2-1
      P[j]       ← (U[j] + ω · V[j])/2
      P[j+n/2] ← (U[j] - ω · V[j])/2
      ω ← ω · ω_n
  return P[0..n-1]
```

- Those / 2's are there because we only divided by (n / 2) during the recursive calls.

Fast Polynomial Multiplication

- Finally, how do we do the full job of multiplying two polynomials in coefficient representation?
- We pad them with enough 0's to make sure we have a power of 2 for the number of coefficients and to make sure we have enough samples for the multiplication. Then we do an FFT to both, multiply their sample representations, and invert the result.

```
FFTMULTIPLY(P[0..m-1], Q[0..n-1]):
  for j ← m to m+n-1
      P[j] ← 0
  for j ← n to m+n-1
      Q[j] ← 0
  P* ← FFT(P)
  Q* ← FFT(Q)
  for j ← 0 to 2^ℓ - 1
      R*[j] ← P*[j] · Q*[j]
  return INVERSEFFT(R*)
```

- Now, the recursive algorithm I showed is not the only way to implement FFTs. The texts go into more detail. In particular, they are often implemented in hardware as circuits with very pretty diagrams.
- Also, these ideas are not exclusively used for multiplying polynomials. In particular, they're most often used now for signal processing as they let you go from a sample based representation of signals to one based on the weighted sum of simple functions like sinusoids.

Selection

- So let's finish up the divide-and-conquer section of the course with an example that has a more complicated recursive structure.
- We're going to look at the problem of element selection. For this problem, we're given an array A[1 .. n] and an integer k where 1 ≤ k ≤ n.
- The *rank* of an element is its index after sorting the array. The rank 1 element is the smallest. The rank n element is the largest. The median element has rank n/2.
- We want to find the element of rank k in A.
- One way to solve Selection is to essentially perform a binary-search like procedure called QuickSelect.
- If you're family with QuickSort, it looks very similar. We pick a *pivot element* and then call this Partition procedure. Partition(A[1 .. n], p) moves the elements of A around so those less than A[p] come before A[p] and those greater than A[p] come later. It then returns the new index of the pivot (i.e., its rank). I'm not going to focus on Partition today except to say it takes O(n) time. CLRS has the proof of correctness if you're interested.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r−1])     ⟨⟨Recurse!⟩⟩
      QUICKSORT(A[r+1..n])     ⟨⟨Recurse!⟩⟩
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                        ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n−1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[1] ↔ A[ℓ + 1]
  return ℓ + 1
```

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)

      if k < r
          return QUICKSELECT(A[1..r−1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k−r)
      else
          return A[r]
```

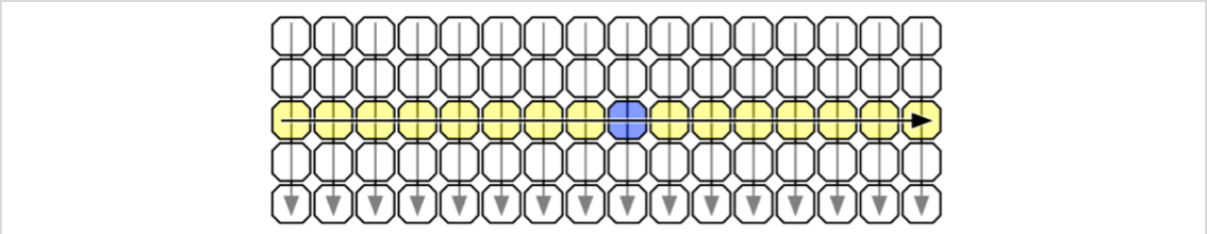
- After partitioning, we can easily determine which half the array contains the rank k element, and do a single recursive call to find it.
- Note how the correctness of the algorithm does not depend on the choice of pivot. Unfortunately, the running time does!
- Let ell be the size of the recursive subproblem we solve. In the worst case, we keep picking

the smallest or largest item as our pivot so that ell = n -1 and the running time is T(n) ≤ T(n-1) + n = O(n^2).

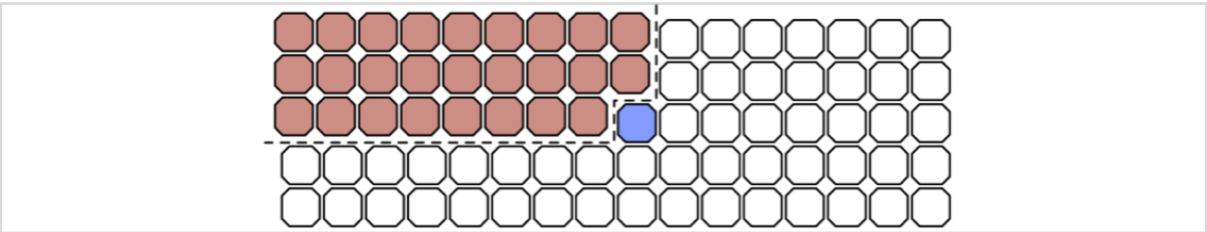
- But! If we somehow choose a pivot closer to the middle so that ell ≤ a n for some constant a < 1, then T(n) ≤ T(a n) + n. The level-sums of the recursion tree form a *decreasing* geometry series, so T(n) = n. If you prefer, we're in the first case of the Master method.
- In the early 1970s, Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan described a way to pick a good pivot for QuickSelect by *recursively* computing the median of a carefully selected and much smaller subset of the input array.
- 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 ceil(n*5) /blocks*. We'll compute the medians of those blocks by "brute force", stick the medians in their own array M, and the use recursion to find the median of M. This "median of medians" will be our pivot. **(In this pseudocode, we're inputing the value of the pivot to partition, not the index.)**

```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MomSelect(M[1..m], ⌊m/2⌋)        ⟨⟨Recursion!⟩⟩

      r ← Partition(A[1..n], mom)

      if k < r
          return MomSelect(A[1..r−1], k)      ⟨⟨Recursion!⟩⟩
      else if k > r
          return MomSelect(A[r+1..n], k−r)    ⟨⟨Recursion!⟩⟩
      else
          return mom
```

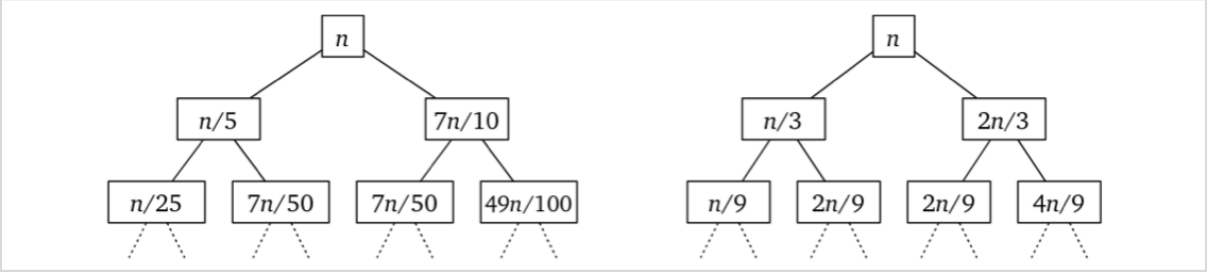
- The algorithm is correct, because it's just a fancy way of picking a pivot for Quickselect, but what is the running time?
- The key insight is that we actually are picking a good pivot as described above.
- To see why, imagine we draw the input array as a 5 x ceil(n/5) grid. Each column represents five consecutive elements from the array.
- However, *for the 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 floor(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) ≤ T(n / 5) + T(7n / 10) + n. We certainly can't use the Master Method in this case!
- But we can still use recursion trees!



- 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).
- 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 * 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) ≤ T(n / 3) + T(2n / 3) + n.
- Now each level of the recursion tree sums to n. There are O(log n) levels, so the running time is O(n log n). We may as well sort the whole array and then grab the kth element.
- Now, having said all that, the constants in the O(n) for MomSelect are pretty big. And in practice, QuickSort is very fast. You should just stick to sorting unless you're working with a

very very large array with millions of elements.

- For the purposes of this course and the QE though, you should remember that you can find the rank k element of an unsorted array in only O(n) time.