CS 6363.003.21S Lecture 5—February 2, 2021

Main topics are #example/quicksort and #example/selection.

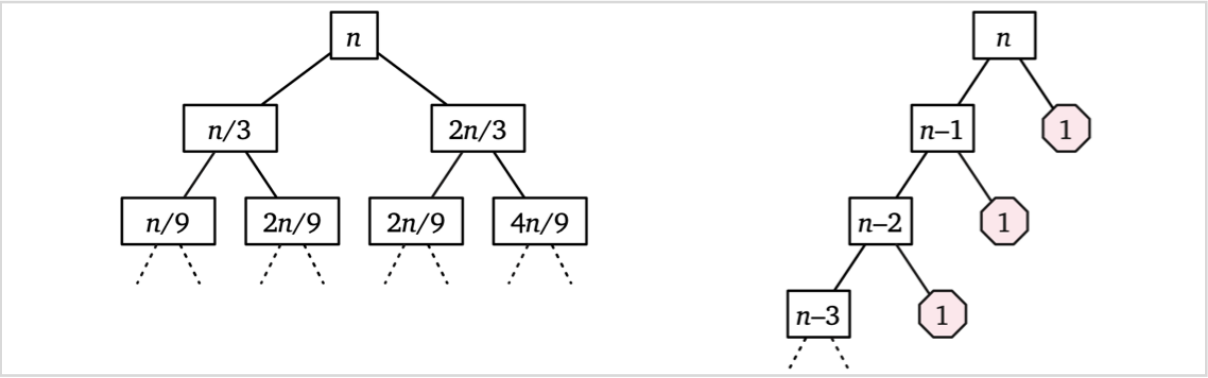
Back to Quicksort

- Last time, we discussed analyzing divide-and-conquer algorithms using recursion trees. We used mergesort as a simple example. So what about quicksort?

```
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[n] ↔ A[ℓ+1]
  return ℓ + 1
```

- Like in mergesort, each subproblem does O(n) work and does two recursive calls, so two children per recursion tree node.
- Unlike mergesort, though, the size of the individual subproblems depend upon r, the *rank* of the pivot element. In particular, the running time is best expressed by the recurrence T(n) = max_{1 ≤ r ≤ n} T(r - 1) + T(n - r) + Theta(n). Since we're picking an arbitrary pivot (maybe the first or last element of the array), all we can guarantee is that the two subproblems have total size n - 1.
- So while we can still guarantee each level i sums to at most n -i, (see figure on right), all we can guarantee about the depth of the tree is that it's at most n.



- In this course, we focus on *worst case* analysis. We want upper bounds that are true no matter what. So we need to accept those pessimistic upper bounds. Quicksort appears to run in O(n^2) time.
- Of course, quicksort seems to run better *in practice*. The intuitive reason is that the pivot tends to be found much closer to the middle of the array, leading to a much more balanced and shallow recursion tree.
- In particular, consider the *median* element, which for this class we'll define as the element

of rank ceil{n / 2}.

- If we pick the median as the pick, then we get a recurrence of T(n) = T(ceil{n / 2} - 1) + T(floor{n / 2}) + Theta(n) ≤ 2T(n / 2) + Theta(n) = Theta(n log n).
- But even if all we can guarantee is that each subproblem has size at most, say, 2n/3, that's enough to have at most log_{3/2} n = O(log n) levels, and a running time of O(n log n).
- A popular heuristic for finding such an element is to use the median of the first, middle, and last array elements as the pivot. Again, this tends to work well in practice, but theoretically, it could still lead to a pivot of very low or high rank, and a worst-case running time of O(n^2).

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, if we try using recursion to solve this problem, 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 ≤ k ≤ 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 described 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.

```
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]
```

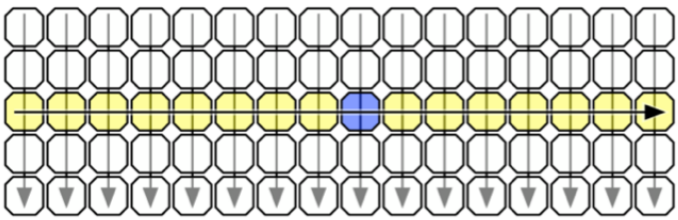
- Notice how we change the second parameter in the second recursive call to take into account that A[r + 1 .. n] is missing the smallest r elements of A[1 .. n].
- Like quicksort, the correctness of the algorithm does not depend on the choice of pivot. Unfortunately, the running time does!
- The worst-case running time follows the recurrence
  - T(n) = max_{1 ≤ r ≤ n} max {T(r - 1), T(n - r)} + Theta(n).
- And like before, we could have r = 1 or r = n, meaning T(n) ≥ T(n - 1) + Theta(n). Again, we have T(n) = Theta(n^2)!
- But! If we somehow choose a pivot closer to the middle so that we recurse on a n elements for some constant a < 1, then T(n) ≤ T(a n) + Theta(n). The level-sums of the recursion tree decrease *exponentially*, so T(n) = O(n).
- So like quicksort, we'd like to find a pivot element that's *close* to the median. Then we can find the exact median in only O(n) time. What we'd really like to see is 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 is disguise!
- 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 their element selection 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 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.
- There's no point in doing recursion when the number of blocks is 5 or less since we need to manually compute medians of five things anyway, so we'll just treat n ≤ 25 as a base case and solve it however seem convenient.
- I'll also assume all elements are distinct (no two are equal). Doing so has no effect on correctness, but it may affect the analysis. In practice, you could implement some kind of tie-breaking rule to enforce distinctness.
- Here's the pseudocode for the whole procedure.

```
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!⟩⟩

      for j ← 1 to n ⟨⟨Find pivot index.⟩⟩
          if A[j] = mom
              p ← j
      r ← Partition(A[1..n], p)

      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
```

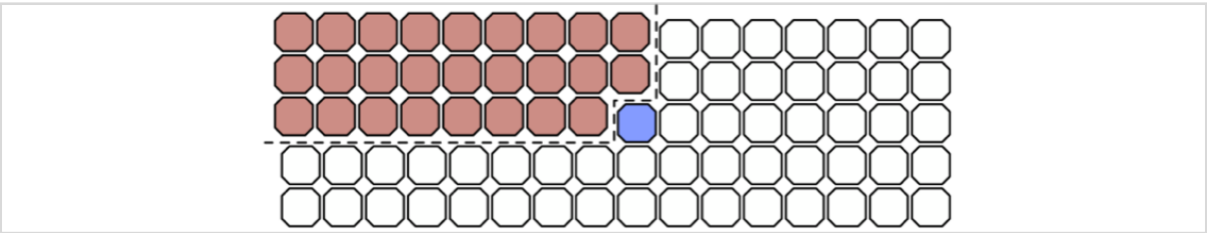
(We're assuming n is a multiple of five here. We can pad the array will infinities or whatever if that's not the case.)

Analysis

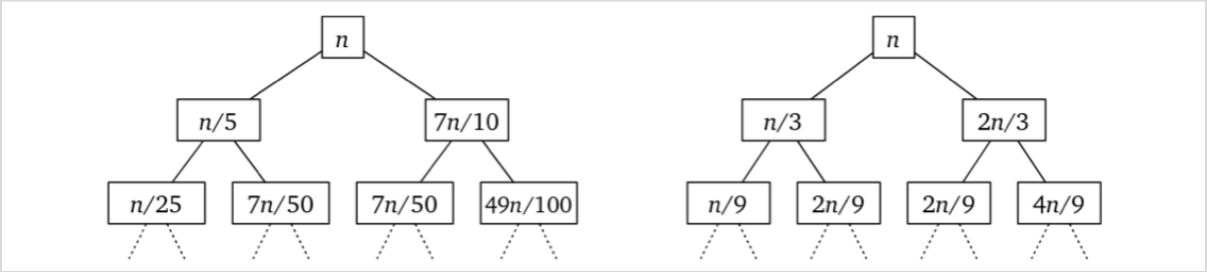
- The algorithm is correct, because it's just a fancy way of picking a pivot for quickselect, but what is the running time?
- The key insight is that we actually are picking a good pivot as described above.
- To see why, imagine we draw the input array as a 5 x ceil(n/5) grid. Each column represents 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 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) + 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 * 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) + O(n).
- 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.

Randomized Quicksort

- But the real best thing to do for both quicksort and quickselect is to pick the pivot element uniformally 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. And a random pivot for quicksort leads to an O(n log n) expected running time!
- Here's a slick analysis of the latter fact. I **do not** expect you to remember this analysis or do anything similar, so don't worry if your probability skills are rusty or you don't follow every

step right away.

- Let E(n) be the *expected* running time of quicksort with a pivot chosen uniformly at random.
- With probability 1/2 we choose a pivot of rank between, say 1/4n + 1 and 3/4n, so
  - E(n) = ≤ 1/2 max_{1/4n + 1 ≤ r ≤ 3/4n} (E(n - r) + E(r - 1)) + 1/2 max_{r not in [1/4n + 1, 3/4n]} (E(n - r) + E(r - 1)) + cn
- E(n) = Omega(n), so it's convex. So E(n - r) + E(r - 1) ≤ E(n) for any r, implying
  - E(n) = ≤ 1/2 max_{1/4n + 1 ≤ r ≤ 3/4n} (E(n - r) + E(r - 1)) + 1/2 E(n) + cn.
- Therefore,
  - 1/2 E(n) ≤ 1/2 max_{1/4n ≤ r ≤ 3/4n} (E(n - r) + E(r - 1)) + cn
- and
  - E(n) ≤ max_{1/4n ≤ r ≤ 3/4n} (E(n - r) + E(r - 1)) + 2cn
- And now we can use recursion trees to argue E(n) ≤ 2cn * log_{4/3} n = O(n log n). Voila!
- With some more cleverness (and by avoiding recurrences), we can actually compute the expected number of comparisons *exactly*. See Erickson's notes on randomized algorithms if you're interested.