CS 4349 Lecture—September 6th, 2017

Main topics for #lecture include #divide-and-conquer, #quicksort, and #selection.

Prelude

- We have a TA! Jon Crain will hold office hours in ECSS 2.104A Tuesdays from 2pm to 3pm.
- Last Wednesday, we discussed solving recurrences using transformations, recursion trees, and the Master Theorem. Any questions?
- This week we'll discuss two final examples of divide-and-conquer and put some of those techniques to good use.

Quicksort

- We'll discuss what should be our last sorting algorithm of the semester.
- Quicksort follows the standard divide-and-conquer paradigm. We choose a pivot element from the array and then
  - Divide the array by placing a subarray of smaller elements than the pivot to the left and placing a subarray of larger elements to the right.
  - Conquer the smaller and larger subarrays by delegating to recursive calls of Quicksort.
  - Combining the solution by doing nothing; the array is already sorted!
- So what would this look like?
  - 24 1 63 97 88 7 . 84 64 75 49 82 **65**
  - 24 1 63 7 64 49 **65** 97 88 84 75 82
  - 1 7 24 49 63 64 **65** 75 82 84 88 97
- Here's what the algorithm looks like in pseudocode:
- QuickSort(A[1 .. n]):
  - if (n > 1):
    - Choose a pivot element A[p]
    - r ← Partition(A, p) partition and return the new index for A[p]
    - QuickSort(A[1 .. r - 1])
    - QuickSort(A[r+1 .. n])
- the index of an element in a sorted array is known as its *rank*, thus the choice of r
- Partition is a subroutine that partitions the array by placing smaller elements to the left of A[p] and larger ones to the right of A[p]
- Partition(A[1 .. n], p):
  - swap A[p] <-> A[n]

    - i ← 0
    - j ← n
    - while (i < j)
        - do i ← i + 1 while (i ≥_ j or A[i] ≥_ A[n])
        - do j ← j - 1 while (i ≥_ j or A[j] ≤ A[n])
        - if (i < j)
            - swap A[i] <-> A[j]
    - swap A[i] <-> A[n}
    - return i
- The proof for partition is ugly, and I'll spare you. The proof for quick sort is simple. If n = 1 you are correct to do nothing. Otherwise, the recursive calls are correct by induction, and sorted smaller + A[r] + sorted bigger is a sorted array.
- Partition takes O(n) time. j - i = n initially, and j - i = 0 at the end. You do a constant amount of work per increment to i or decrement to j.
- Let T(n) be the running time of Quicksort given our pivot has rank r. We have T(n) = T(r - 1) + T(n - r) + O(n).
- Ideally, we'd have r = \floor{n/2}, i.e., we pivot on the median element. In this case, T(n) ≤_ 2T(n/2) + O(n). Which we've seen a few times comes out to T(n) = O(n log n).
- Unfortunately, it's not easy to find a pivot of rank n/2. So most programmers tend to do something easier like use A[1] or A[n] as the pivot. But then you risk grabbing something of rank 1 or n, giving you the recurrence T(n) = T(n-1) + O(n) = \sum_{i=1}^n O(i) = O(n^2).
- There are other heuristics for picking a pivot. The most popular, other than picking the first or last element, is probably the "median-of-three" heuristic. Pick three elements, probably A[1], A[n/2], and A[n], and pivot on the median of these three values. It tends to work well in practice, but you can still have inputs where T(n) = T(1) + T(n-2) + O(n) = O(n^2).
- But why does it work well in practice? Because intuitively, the median of three does have rank somewhere near n / 2, maybe between, say, n / 10 and 9n / 10. If you could guarantee the pivot's rank was always in that range, then T(n) = T(n / 10) + T(9n / 10) + O(n). **[recursion tree]** You get T(n) = O(n log n).
- So Quicksort is an example of a divide-and-conquer algorithm that works well in practice, and there is some theory as to why, even though it runs as slowly as Insertion-Sort in the worst case.
- It's also a good explanation for why you should go for balanced subproblems when designing divide-and-conquer algorithms.

Selection

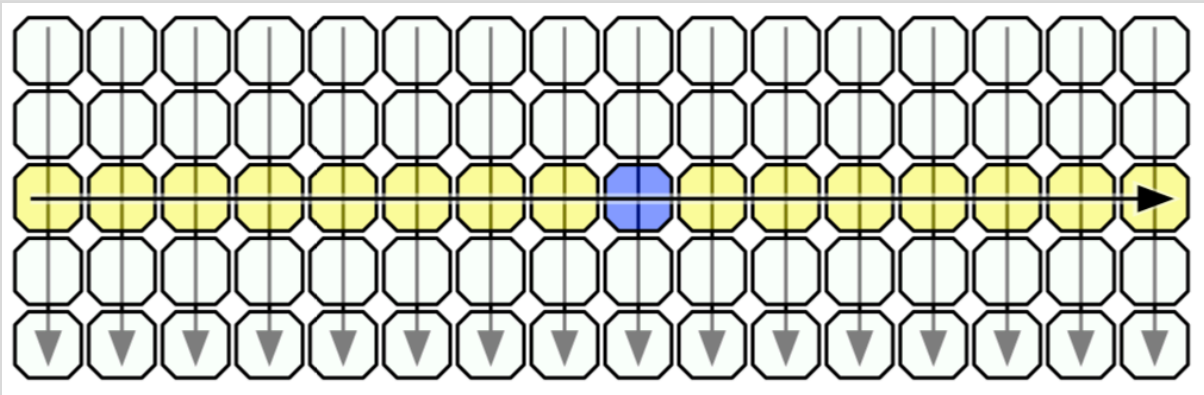
- But what if you really do want to find the median element.
- Given an array A[1 .. n] and an integer k ≥_ 1, the Selection problem asks for the element of

rank k in A (or A[n] if n < k).

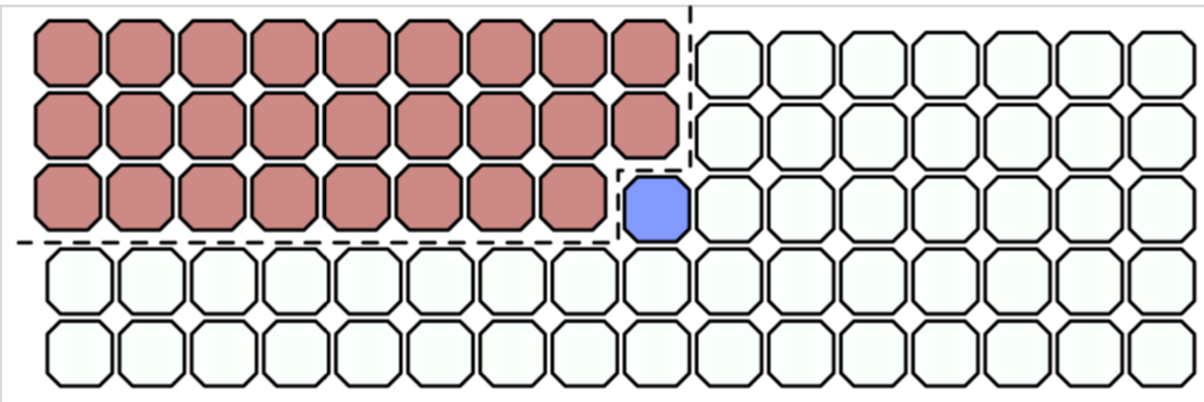
- One way to solve Selection is to do a “one-armed quick sort” known as Quickselect.
- QuickSelect(A[1 .. n], k):
  - if n = 1
    - return A[1]
  - else
    - Choose a pivot element A[p]
    - r ← Partition(A[1 .. n], p)
    - if k < r
      - return QuickSelect(A[1 .. r-1], k)
    - else if k > r
      - return QuickSelect(A[r+1 .. n], k - r)
    - else
      - return A[r]
- Again, the running time depends upon our choice of pivot. Let ell be the size of the recursive subproblem. 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) + O(n) = O(n^2).
- But! If we choose a pivot closer to the middle so that ell ≤ a n for some constant a < 1, then T(n) ≤ T(a n) + O(n). By Master Theorem or recursion trees, T(n) = O(n).
- So what we’ll do is choose a guaranteed good pivot, by *recursively* finding the median of a guaranteed smaller subset of the input array.
- This smaller set will be the median of medians, or MoM.
- For convenience, let A[n + c] = infinity
- MoMSelect will rearrange A and then return the index of the rank k element after A has been rearranged. The solution to the Selection problem is therefore A[MoMSelect(A[1 .. n], k)].
- MoMSelect(A[1 .. n], k):
  - if n ≤ 25
    - use brute force
  - else
    - m ← \ceil{n/5}
    - M ← array of length m
    - for i ← 1 to m
      - M[i] ← Median(A[5i - 4 .. 5i]) select median of five elements
    - mom ← MoMSelect(M, \floor{m / 2})

    - r ← Partition(A[1 .. n], mom)
    - if k < r
      - return MoMSelect(A[1 .. r-1], k)

    - else if k > r
        - return MoMSelect(A[r+1 .. n], k - r)
    - else
        - return mom
- So we're recursively calling MoMSelect on the medians of the five element subarrays. If T(n) is the running time of MoMSelect, that call takes T(n / 5) time.
- But what about the second call?



- Imagine drew the array as a 5 x \ceil{n / 5} grid so each column was five consecutive elements. Then we sort every column **independently** from top down, and then sort the columns by their median elements from left to right.
- The algorithm does not do this!
- So here's the median of these medians, right in the middle.
- There are \ceil{\ceil{n / 5} / 2} - 1 lessor medians to its left. That's about n/10. And each of those lessor medians is at least as large as 3 elements from their column. So, the median of medians is larger than about 3 * n / 10 elements.



- k > r, then those 3n / 10 elements do not appear in the second recursive call. That call takes T(7n / 10) time.
- Symmetrically, there are about 3n / 10 elements that are bigger than the median of medians. If k < r, then those elements don't appear in the recursive call which again takes T(7n / 10) time.
- So T(n) ≤_ T(n / 5) + T(7n / 10) + cn for some constant c.
- We'll use recursion trees.
- The root gets cn. The row at depth i sums to (9 / 10)^i * cn.

- The full rows are decreasing geometric series, and the less than full rows sum to values even smaller. The largest term dominates so $T(n) = O(n)$.
- That said, this is not a practical algorithm unless n is several million. Otherwise, you may as well sort the array in $O(n \log n)$ time and then pick the kth element. Or just run QuickSelect using some reasonable heuristic for the pivot and hope for the best.
- Now, there is a way to pick pivots that is both practical and theoretically sound, though. Pick an element of A uniformly at random. No matter what A is, the *expected* running time of Quicksort will be $O(n \log n)$, the expected running time of QuickSelect will be $O(n)$, and the time to pick a pivot will be barely worse than running one of the common heuristics. We might come back to this later in the semester.