Divide and Conquer 2

Sam McCauley

March 11, 2024

Welcome Back!

- Assignment due Wednesday
- Any questions?

Geometric Sums

Geometric Sum

A *geometric sum* is of the form $\sum_{i=0}^{k} r^i$. They come up frequently in computer science (and elsewhere). We have that, for any $r \neq 1$,

$$\sum_{i=0}^{k} r^i = \frac{1 - r^{k+1}}{1 - r}$$

Proof: Here's a clever way to solve this sum. We'll see a similar technique when we get to randomized algorithms later in the class.

Let $S = \sum_{i=0}^{k} r^i$. Then:

$$r \cdot S = r \sum_{i=0}^{k} r^i = \sum_{i=1}^{k+1} r^i = r^{k+1} - 1 + \sum_{i=0}^{k} r^i.$$

In other words, $rS = (r^{k+1} - 1) + S$. Solving, $S = (1 - r^{k+1})/(1 - r)$.

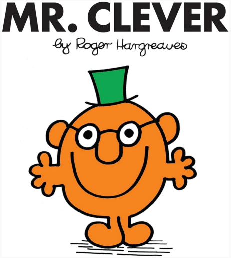
Divide and Conquer Multiplication

Divide and Conquer: Multiplication

$$a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$$

- To multiply two n-digit numbers, we first perform four recursive multiplications:
    - $a_\ell \times b_\ell$, $a_\ell \times b_r$, $b_\ell \times a_r$, and $a_r \times b_r$
- And then we add them together (and multiply by 10^n) in $O(n)$ time.
- If $n = 1$ just multiply the numbers
- Recurrence?
- $T(n) = 4T(n/2) + O(n)$; $T(1) = 1$
- Get $\Theta(n^2)$ time, same as before. *Can we improve this?*

Divide and Conquer: Karatsuba's Algorithm



$$a \times b = 10^n(a_\ell b_\ell) + 10^{n/2}(a_\ell b_r + b_\ell a_r) + a_r b_r$$

- Consider the following *three* recursive multiplications
  - $(a_\ell \times b_\ell)$, $(a_r \times b_r)$, and $(a_\ell + a_r) \times (b_\ell + b_r)$
- I claim this is enough! Why?
- $a_\ell b_r + b_\ell a_r = (a_\ell + a_r) \times (b_\ell + b_r) - a_\ell \times b_\ell - a_r \times b_r$
- So after *three* recursive calls of size $n/2$ I can calculate $a \times b$. I used $O(n)$ total time other than the recursive calls
- $T(n) = 3T(n/2) + O(n)$; $T(1) = 1$

Solving the Multiplication Recurrence

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = 1$$

- Let's solve this recurrence [On Board #1]
- We want to ask ourselves: What is the height of the tree? What is the cost of each level?
- Solution: $O(n^{\log_2 3}) = O(n^{1.58})$ time
- Much better than n^2!
- Reflect: why did changing a *constant* from 3 to 4 have such an impact on the running time?

Multiplying Numbers Efficiently

- Kolmogorov conjectured that $\Omega(n^2)$ time is needed; stated this conjecture in a seminar at Moscow State University in 1960
- Karatsuba, a student figured out this $O(n^{\log_2 3})$ time algorithm in the next week
- Kolmogorov cancelled the whole seminar and then published the result on Karatsuba's behalf without telling him
- Can we do better?
- Best known: $O(n \log n)$ [Harvey, van der Hoeven 2019]
- Are these speedups useful in practice?
    - Sometimes! Karatsuba's is used in some libraries

More Recurrences

Divide and Conquer and Recurrences

- We analyze divide and conquer algorithms using *recurrences*
- Gives us a bird's eye view of the cost of the algorithm
- Recurrence relations can also guide us in searching for algorithms
  - "How can I sort in $O(n \log n)$ time?"
  - If my sorting method recurses on two halves, and does $O(n)$ additional work, I get $T(n) = 2T(n/2) + O(n)$, which gives $O(n \log n)$
  - (Of course, this is just a starting point: many other recurrences solve to $O(n \log n)$.)
- Let's look at some other recurrences

Three practice recurrences

Let's do the following recurrences [On Board #2]

For all of these assume $T(1) = 1$.

$$T(n) = 4T(n/2) + O(1)$$

$$T(n) = 2T(n/2) + O(n \log n)$$

$$T(n) = 3T(n/3) + O(n)$$

On Floors and Ceilings in Recurrences

- Most input sizes are not (say) powers of 2
- Merge sort's actual recurrence is:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$$

- Does this change the solution?
- No. *We will ignore all floors and ceilings in this class.* See Erikson 1.7 for some formal justification

Tree Height and Recurrences that Don't Branch

Let's do the following recurrences [On Board #3]

$$T(n) = T(n/2) + O(1)$$

$$T(n) = T(\sqrt{n}) + O(1)$$

$$T(n) = T(n/2) + O(n)$$

Three kinds of recurrences

Recurrences often fit into one of three types:

- Cost at the root dominates
- Cost at the leaves dominate
- Cost at each level is the same

Ways to Solve Recurrences

- Recursion tree (recommended)
- Guess and check
  - If we have the solution for $T(n)$, we can substitute it into the recurrence to check that it is satisfied
  - Can formalize using induction
  - “Unroll” recurrence a few steps to get intuition before guessing
- Master theorem (next slide) *gives* the solution for many common recurrences

Master Theorem (Simple Version)

For *constants* a and b and a function $f(n)$, to solve

$$T(n) = aT(n/b) + f(n); \qquad T(1) = 1$$

- If $f(n) = O(n^c)$ for $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$
  - So $T(n) = 4T(n/2) + O(n)$ solves to $T(n) = \Theta(n^2)$
- If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
  - So $T(n) = 2T(n/2) + O(n)$ solves to $T(n) = \Theta(n \log n)$
- A fast way to solve simpler recurrences. But a pain to memorize and only works situationally.

Binary Search

Binary Search

```
binary_search(key, A, start, end):
    mid = (start + end)/2
    if key == A[mid]:
        return mid
    else if key < A[mid]:
        return binary_search(key, A, start, mid-1)
    else:
        return binary_search(key, A, mid+1, end)
```

- Correctness intuition: we recurse on the half of A that must contain key.
- How would we prove correctness formally?
- Running time? $T(n) = T(n/2) + O(1)$ We've seen: $T(n) = O(\log n)$

Binary Search on a Linked List?

This is *not a good algorithm*. But I’ve seen people implement it many times.

Today: how efficient is it?

We can binary search by:

- Find the middle item of the linked list
    - By iterating through the linked list
- Compare to query item
- Recurse on first or second half of the linked list
- Recurrence?
- $T(n) = T(n/2) + \Theta(n)$
- Solution: $\Theta(n)$ time
- (Could have just scanned!)

Selection

Median finding

- Goal: given an *unsorted* array A of length n, find the median of A
- Can someone give an $O(n \log n)$ time algorithm to solve this?
- Sort A using Merge Sort. Return $A[\lceil n/2 \rceil]$
- Can we do better?

Linear-Time Median Finding

- Goal: an $O(n)$ algorithm to find the median of any unsorted array A
- Can't sort! Is it really possible to find the median of an array without sorting it?
- We'll solve a more general problem: find the kth largest element in the array
- Divide and conquer algorithm; invested by Blum, Floyd, Pratt, Rivest, Tarjan 1973

Partition (Selection Subroutine)

```
Partition($A$, $p$):
    Create empty arrays $A_{<p}$ and $A_{>p}$
    for i = 0 to $|A| - 1$:
        if $A[i] < p$:
            add $A[i]$ to $A_{<p}$
        if $A[i] > p$:
            add $A[i]$ to $A_{>p}$
    return $|A_{<p}|, A_{<p}, A_{>p}$
```

Returns two arrays, one with elements $< p$ and one with elements $> p$

The *rank* of p is the number of elements in A smaller than p; also returns the rank of p.

Selection (First Attempt)

```
Select($A$, $k$):
    if $|A| = 1$:
        return $A[0]$
    else:
        choose a pivot $p$ # we'll define how later
        $r, A_{<p}, A_{>p}$ = Partition($A$, $p$)
        if k == r:
            return $p$
        else:
            if $k < r$:
                return Select($A_{<p}, k$)
            else:
                return Select($A_{>p}, k - r - 1$)
```

The main question is: How do we select our pivot? (And how does that impact performance?)

How good does our pivot selection need to be?

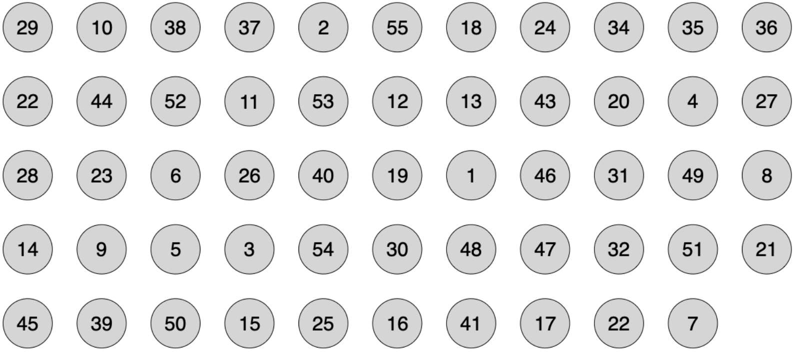
- Let’s say our pivot is *not* in the first or last $3n/10$ items of A (where $n = |A|$)
- What is our recurrence?
- $T(n) \leq T(7n/10) + O(n)$
- $T(n) = O(n)$

Finding the Pivot: Goal

- Find a pivot that has rank between $3n/10$ and $7n/10$ in time $O(n)$
- The array is *unsorted*
- Want to always be successful
- Note: Can verify in $O(n)$ time!

Finding an Approximate Median

- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group

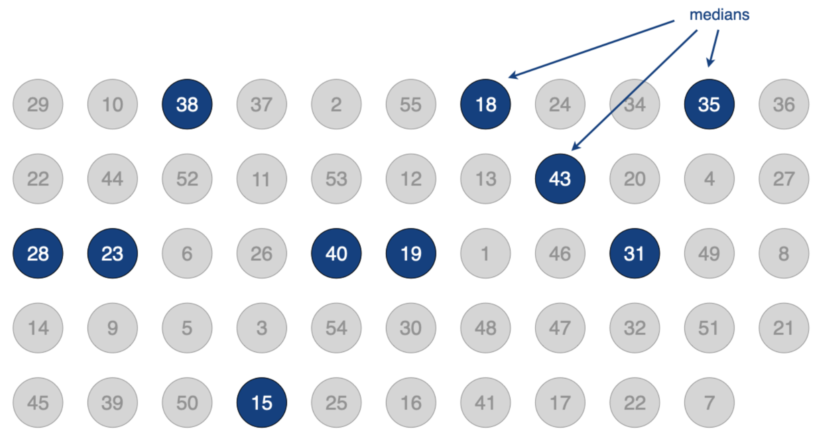


29	10	38	37	2	55	18	24	34	35	36
22	44	52	11	53	12	13	43	20	4	27
28	23	6	26	40	19	1	46	31	49	8
14	9	5	3	54	30	48	47	32	51	21
45	39	50	15	25	16	41	17	22	7	

n = 54

Finding an Approximate Median

- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group

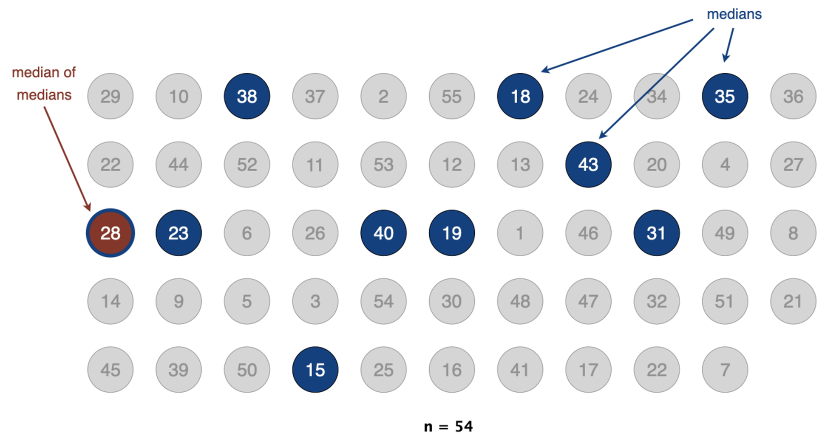


29	10	**38**	37	2	55	**18**	24	34	**35**	36
22	44	52	11	53	12	13	**43**	20	4	27
28	**23**	6	26	**40**	**19**	1	46	**31**	49	8
14	9	5	3	54	30	48	47	32	51	21
45	39	50	**15**	25	16	41	17	22	7	

n = 54

Finding an Approximate Median

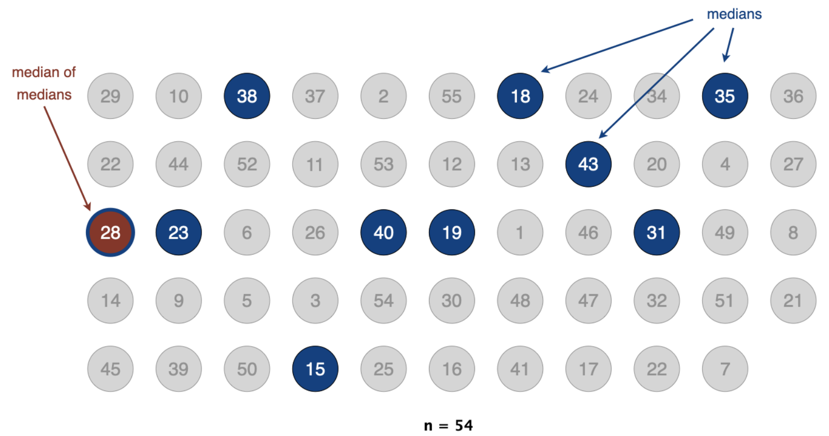
- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find the median of these $\lceil n/5 \rceil$ medians; this is our pivot



Finding an Approximate Median

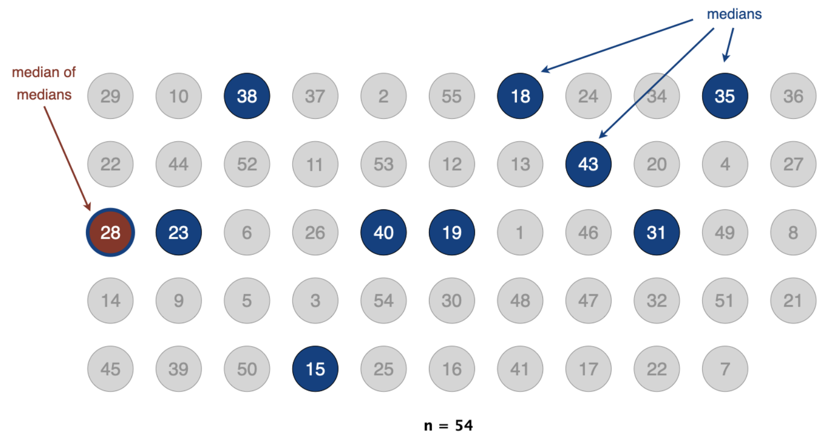
- Divide the array into $\lceil n/5 \rceil$ groups of 5 elements (ignore leftovers)
- Find median of each group
- Find the median of these $\lceil n/5 \rceil$ medians; this is our pivot (call it M)
- How can we find the median of these medians? Recursively!
  - This is a median-finding algorithm! We call `Select` to find the median of these medians to get our pivot

Rank of the Median of Medians



- What elements are smaller than the median of medians M?
- Half the medians ($n/10$ elements)
- Also: for each such median, two elements the median's list ($2n/10$ elements)

Rank of the Median of Medians



- $\geq 3n/10$ are less than M
- Similarly: $\geq 3n/10$ are greater than M
- So M is a good pivot!

Linear-Time Selection

```
Select($A$, $k$):
    if $|A| \leq 5$:
        return $k$th largest element of $A$
    else:
        divide $A$ into $\lceil n/5 \rceil$ groups of 5 elements
        Create array $A_m$ containing the median of each group
        $p$ = Select($A_m, \lceil |A_m|/2 \rceil$)
        $r, A_{<p}, A_{>p}$ = Partition($A$, $p$)
        if k == r:
            return $p$
        else:
            if $k < r$:
                return Select($A_{<p}, k$)
            else:
                return Select($A_{>p}, k - r - 1$)
```

Recurrence: $T(n) = T(n/5) + T(7n/10) + O(n)$; $T(5) = O(1)$ [On Board #4]

Median Finding

- An advanced Divide and Conquer application
- Uses a nontrivial recurrence
- Can find median of an unsorted array in $O(n)$ time—strictly faster than sorting

How fast can we sort?

- A *comparison-based* sorting algorithm has no assumptions on the elements we are sorting
- I don't know whether a given element is likely to be "big" or "small"
- All I can do is compare two elements: is $A[i] \le A[j]$?
- Insertion sort, selection sort, merge sort, quicksort, etc., are all comparison-based
- (Can do better than comparison-based for some special cases, e.g. if all numbers in the array are from $\{1, 2, \ldots, n\}$. But comparison-based sort is how we sort items in general.)

Lower Bound

Theorem

Any comparison-based sorting algorithm makes $\Omega(n \log n)$ comparisons in the worst case.

Proof: Consider a comparison-based sorting algorithm that makes k comparisons. There are 2 outcomes to every comparison ($A[i] \leq A[j]$ is true or false); so there are 2^k possible outcomes to this sorting algorithm.

Any sorting algorithm needs to correctly sort any permutation of the n items. There are $n!$ permuations of the items. So we need $2^k > n!$.

First, we lower bound (assume n is even for simplicity):

$$n! = n(n-1)(n-2)\ldots(n/2+1)(n/2)(n/2-1)\ldots 1 n! \quad \geq (n/2)(n/2)(n/2)\ldots(n/2)(n/2)$$

So $2^k > n! > (n/2)^{n/2}$. Taking logs of both sides, $k > \frac{n}{2}\log_2 \frac{n}{2} = \Omega(n \log n)$.