Algorithm Design and Analysis

Introduction, Algorithm Analysis, and Selection

Formal analysis of algorithms

- We want **provable guarantees** about the properties of algorithms
  - E.g., **prove** that it runs in a certain amount of **time**
  - E.g., **prove** that it outputs the correct **answer**
- **Important question:** How exactly do we measure time?
  - **Answer**: It depends :)
  - Lots more discussion about this in the coming lectures
- We need a ***model of computation!***
  - Specifies exactly what operations are permitted
  - How much each operation costs (sometimes called the *cost model*)

Today’s model

- ***The Comparison Model***
  - The initial input to the algorithm consists of an array of n comparable elements in some initial order
  - The algorithm may perform comparisons (ask is $a_i < a_j$?) at a **cost of 1**
  - Copying/swapping/moving elements is *free*
  - The elements **can not** be assumed to be integers, numbers, strings, tuples of those, or any specific type

Quicksort: A journey of algorithm design and analysis

- As seen in 15-122 and 15-210 (and possibly elsewhere!)
- One of the most well-known algorithms in all of computer science

function quicksort($\boldsymbol{a}[\mathbf{0} \ldots \boldsymbol{n}-\mathbf{1}]$ **: list) {**
  select a pivot element p = a_i for a some i
  let LESS = $[a_j$ such that $a_j < p]$
  let GREATER = $[a_j$ such that $a_j > p]$
  return quicksort(LESS) + $[p]$ + **quicksort**(GREATER)
}

Question: What is the ***complexity*** of Quicksort?

Which measure of complexity?

Definition (Worst-case complexity): The worst-case complexity of an algorithm is the *largest cost* it can incur over *any possible input* (usually as a function of input size n)

Theorem (15-122): The *worst-case* cost of QuickSort on an input of length n *is* $O(n^2)$

Which measure of complexity?

Definition (Average-case complexity): The average-case complexity of an algorithm is the *average of the costs* of the algorithm over *every possible input.*

Note: Mathematically, this is equivalent to the *expected value of the cost* of the algorithm over an *input chosen uniformly randomly*.

Theorem (15-210): The *average-case* cost of QuickSort on an input of length n *is* $O(n \log n)$

Making it better

- The average-case performance of QuickSort is great
- But its only reliable if the input is random! An evil adversary can always feed our code a worst-case input and ruin our day :(
- Most real-life data **is not random**. Hoping that data is random is not a good way to design your algorithms.

Important idea: Instead of hoping that the input is random… put the randomness *into the algorithm*!

Making it better: *Randomized* Quicksort

function random_quicksort($\boldsymbol{a[0 \dots n-1]}$ **: list) {**
    select a ***random*** pivot element p = a_i
    let LESS = $[a_j$ such that $a_j < p]$
    let GREATER = $[a_j$ such that $a_j > p]$
    return random_quicksort(LESS) + $[p]$ + **random_quicksort**(GREATER)
}

Analyzing randomized algorithms

Theorem (15-210): The *expected* number of comparisons performed by randomized Quicksort on an input of size n *is at most* $O(n \log n)$

IMPORTANT NOTES:

Note: When analyzing randomized algorithms, we are usually interested in the ***expected value*** *over the random choices* to process a ***worst-case user input***

- we are **not** assuming that our random-number generator gives us the worst possible random numbers,
- we are **not** analyzing the algorithm for a randomly chosen input (that's average-case complexity!)

The Quicksort journey so far

Its fast in practice!

Worst-case cost is $O(n^2)$

Average-case cost is $O(n \log n)$

Randomized Quicksort costs $O(n \log n)$ *in expectation*

Deterministic Quicksort in worst-case $O(n \log n)$ cost??

What if we could efficiently find the median element and use that as the pivot?

New problem: Median / k^{th} smallest

Problem (Median) Given a range of distinct elements $a_1, a_2, \dots, a_n$, output the median.

Definition (Median) The median is the element such that exactly $\lfloor n/2 \rfloor$ elements are larger

- More generally, we can try to solve the "k^{th} smallest" problem. Given a range of distinct elements and an integer k, we want to find the element such that there are exactly k smaller elements

Algorithm design strategy

Algorithm design idea: Start with a simple but inefficient algorithm, then optimize and remove unnecessary steps.

Simple algorithm (k^{th} smallest): Sort the array and output element k

- **Redundancy**: We are finding the k^{th} *smallest* for **every** k

Take inspiration from Quicksort?

```
function quicksort(a[0 … n − 1]) {
    select a pivot element p = a_i
    let LESS = [a_j such that a_j < p]
    let GREATER = [a_j such that a_j > p]
    return quicksort(LESS) + [p] + quicksort(GREATER)
}
```

Question: If we only want the k^{th} number, what is wasteful here?

return $\underbrace{\textbf{quicksort}(\text{LESS})} + [p] + \underbrace{\textbf{quicksort}(\text{GREATER})}$

The answer is either in here

Or the answer is in here

The result: Randomized Quickselect

function quickselect($a[0 \dots n-1],\ k$**) {**
select a random pivot element p = a_i for a random i
let LESS = [a_j such that $a_j < p$]
let GREATER = [a_j such that $a_j > p$]

if k > |LESS| **then return** quickselect(GREATER, k-|LESS|-1)

else if k<|LESS| **then return** quickselect(LESS, k)

else return p
}

Now the analysis

Theorem: The *expected* number of comparisons performed by Quickselect on an input of size n *is at most* $8n$

Warning: The proof is subtle because it uses probability. We must be careful to not make false assumptions about how probability and randomness work…

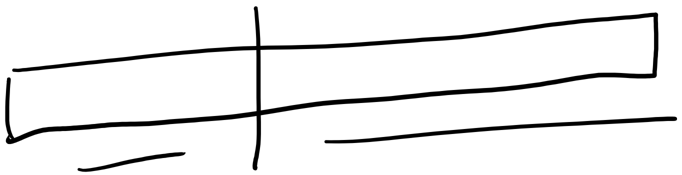
Let $T(n) =$ the ***expected*** number of comparisons performed by Quickselect on a ***worst-case input*** of size n

$$T(n) = n - 1 + \mathbb{E}_X[T(X)]$$

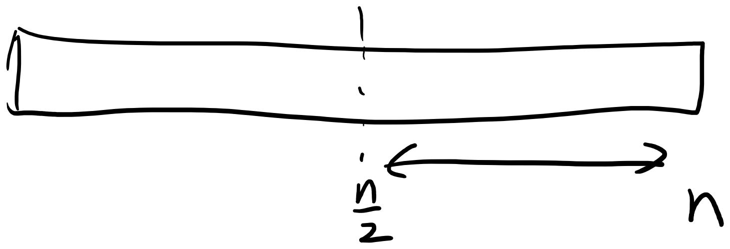
X = size of subproblem!

First attempt: Almost-correct analysis

Note: This proof is nearly, but not quite correct. It does, however, provide some useful insight that gets us closer to a correct proof.



$\frac{3n}{4}$

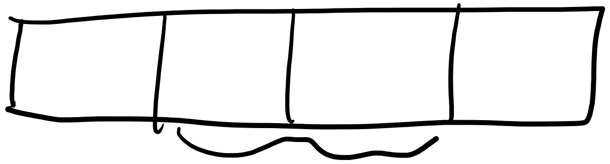


$$T(n) \leq n-1 + \underbrace{T\left(\tfrac{3}{4}n\right)}$$

$$T(E[X]) \neq E[T(X)]$$

A better proof

Question: Let's be more precise. How often is the recursive subproblem size at most $\underline{3n/4}$?



$1/2$

So, a better recurrence relation is

$$T(n) \le n - 1 + \frac{1}{2} T\left(\tfrac{3}{4} n\right) + \frac{1}{2} T(n)$$

$$\le 2(n-1) + T\left(\tfrac{3}{4} n\right)$$

Validating the recurrence relation

$$T(n) \le 2(n-1) + T(3n/4)$$

$$T(n) \le 2(n-1) + 8 \cdot \frac{3n}{4}$$

$$= 2(n-1) + 6n$$

$$< 8n$$

$\square$

Summary of randomized Quickselect

```
function quickselect(a[0 … n − 1], k) {
    select a random pivot element p = a_i for a random i
    let LESS = [a_j such that a_j < p]
    let GREATER = [a_j such that a_j > p]

    if |LESS| > k then return quickselect(LESS, k)
    else if |LESS| = k then return p
    else return quickselect(RIGHT, k − |LESS| − 1)
}
```

- Runs in $O(n)$ **expected** time in the **comparison model**.
- More tightly, uses at most $8n$ comparisons in expectation.
- An as exercise, the analysis can be improved to $4n$ comparisons.

Have we achieved our goal?

- Use Quickselect to select the pivot for Quicksort
- Guaranteed best-case recursion for Quicksort
- Problem?

- Randomized Quickselect is still… randomized.
- So, Quicksort would still be $O(n \log n)$ randomized, not deterministic

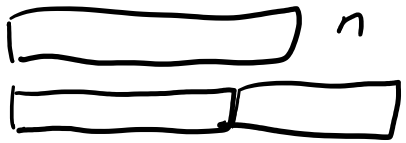
We need a deterministic algorithm!!

- Where was the randomness in Randomized QuickSelect? How can we get rid of it?

- What if we could deterministically find the optimal pivot? What would that be? The median! Oh…

What we need: In $O(n)$ comparisons, we need to find a "good" pivot. A good pivot would leave us with cn elements in the recursive call, for some fraction $c < 1$, e.g., $3n/4$ elements is good.

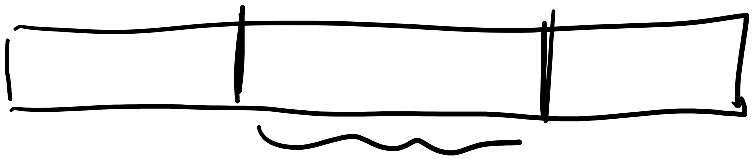
Picking a good pivot



- Picking the median as the pivot is too much to ask for, so we want some kind of "approximate median"

Idea (doesn't quite work, but very close): Pick the median of a smaller subset of the input (faster to find) then hope that it is a good approximation to the true median.

Question: What if we find the median of half of the elements?



$$T(n) \le n-1 + \underbrace{T\left(\tfrac{3}{4}n\right)}_{\text{recurse}} + \underbrace{T\left(\tfrac{n}{2}\right)}_{\text{find pivot}}$$

Median of half

If we pivot on the median of half of the elements, the number of comparisons will be

$$T(n) \leq n-1 + T\left(\frac{3n}{4}\right) + T\left(\frac{n}{2}\right)$$

$$T(n) = O(n^{1.\sim})$$

Exercise: Show that picking any constant-fraction sized subset (e.g., a quarter, one tenth) and taking the median doesn't work.

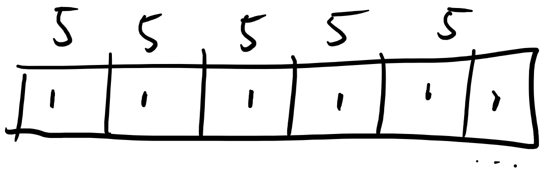
We need to go deeper!

Note: This idea is extremely subtle. It took <u>four</u> Turing Award winners to figure it out. We don't expect that you would produce this algorithm on your own.

- Finding the median of a smaller set **almost** worked, but it was just a bit too much work since the "approximate median" wasn't good enough.

Huge idea (median of medians): Find the medians of several small subsets of the input, then find the **median of those medians**.

Median of medians algorithm



function DeterministicSelect($a[0 \dots n-1],\ k$**) {**

  group the array into $n/5$ groups of size 5, find the median of each group

  recursively find the median of these medians, call it p

  // Below is the same as Randomized Quickselect

  let LESS = $[a_j$ such that $a_j < p]$

  let GREATER = $[a_j$ such that $a_j > p]$

  if $|\mathrm{LESS}| > k$ **then return DeterministicSelect**(LESS, k)

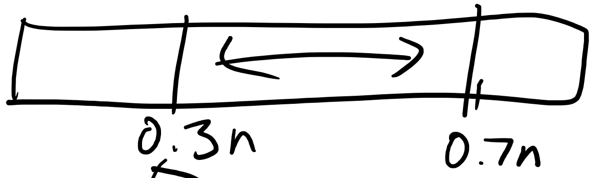
  else if $|\mathrm{LESS}| = k$ **then return** p

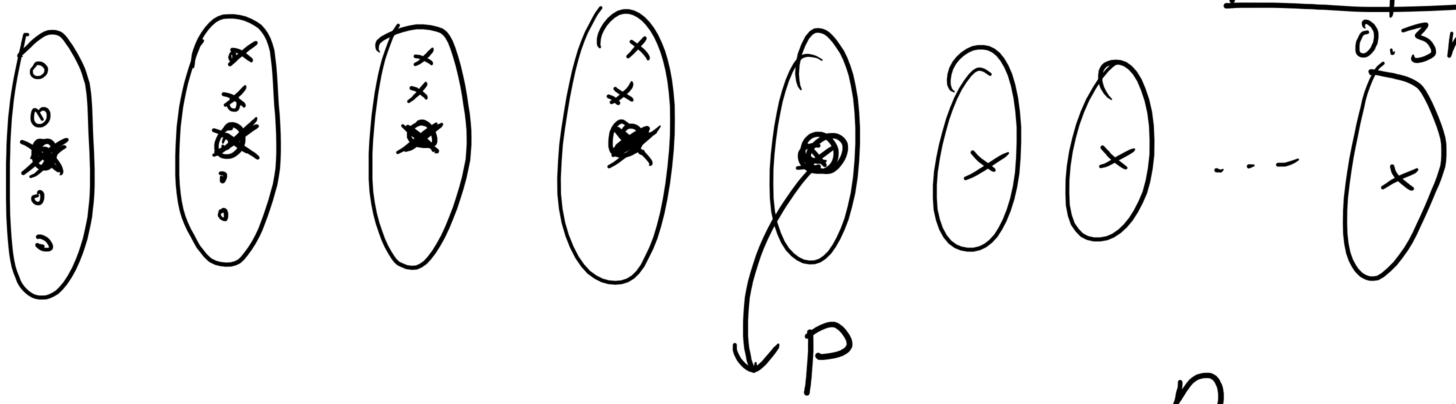
  else return DeterministicSelect(RIGHT, $k - |\mathrm{LESS}| - 1$)

}

How good is the median of medians?

Theorem: The median of medians is larger than at least $3/10^{\text{ths}}$ of the input, and smaller than at least $3/10^{\text{ths}}$ of the input





$$\frac{\frac{n}{5}}{2} + 2 \cdot \frac{n}{10} = \frac{3n}{10}$$

Analysis of DeterministicSelect

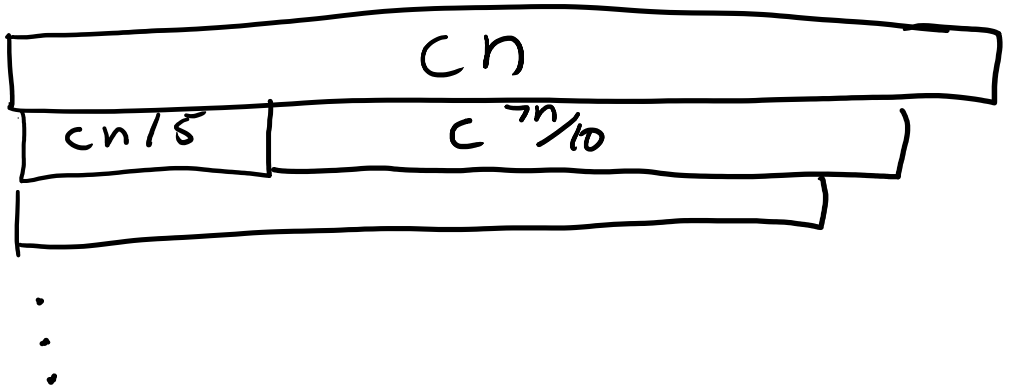
***Theorem*:** The number of comparisons performed by DeterministicSelect on an input of size n is $O(n)$

1. Find the median of $n/5$ groups of size 5 $\quad O(n)$
2. Recursively find the median of medians $\quad T\left(\frac{n}{5}\right)$
3. Split the input into LESS and GREATER $\quad n-1$
4. Recurse on the appropriate piece $\quad T\left(\frac{7n}{10}\right)$

$$T(n) \leq O(n) + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

Solving the recurrence

$$T(n) \leq cn + T(n/5) + T(7n/10)$$



$$cn + c \cdot \frac{9}{10} n + c \cdot \frac{81}{100} n + \ldots$$

So, the total running time is…

$$T(n) \leq cn(1 + (9/10) + (9/10)^2 + (9/10)^3 + \cdots)$$

$r = \frac{9}{10}$ $\qquad \frac{1}{1-r} = \frac{1}{1-\frac{9}{10}} = 10$

$T(n) \leq 10 \cdot cn$

$= O(n)$ $\qquad \square$

Summary of DeterministicSelect

```
function DeterministicSelect(a[0 ... n − 1], k) {
   group the array into n/5 groups of size 5,
      find the median of each group
   recursively find the median of these medians, call it p

   // Below is the same as Randomized Quickselect
   let LESS = [a_j such that a_j < p]
   let GREATER = [a_j such that a_j > p]
   if |LESS| ≥ k then return DeterministicSelect(LESS, k)
   else if |LESS| = k then return p
   else return DeterministicSelect(RIGHT, k − |LESS| − 1)
}
```

- The median of medians is the key ingredient for getting a deterministic algorithm
- To analyze the recurrence, we used the “stack of bricks” method.
- We could also prove it by induction, but this requires us to know the runtime already

The Quicksort journey

Its fast in practice!

Worst-case cost is $O(n^2)$

Average-case cost is $O(n \log n)$

Randomized Quicksort costs $O(n \log n)$ *in expectation*

Deterministic Quicksort in worst-case $O(n \log n)$!!

1. *Use the median-of-medians algorithm to find the median in* ***deterministic*** $O(n)$ *cost*
2. *Use the median as the pivot for Quicksort*

Take-home messages for today

- There’s more to Quicksort than you think!
- Recursion is powerful, randomization is powerful.
- Analyzing ***randomized recursive algorithms*** is tricky. Be careful with expected values!!
- Analyzing runtime via ***recurrence relations*** is very useful.