Binary Search (and variants)

Applicable when with one query, the search space size can be reduced by half. \( N \leq N/2 \) \( \log N \) No. of queries

Classic Example:
Given a sorted integer array \( A \) and an integer \( x \), find the location of \( x \) in \( A \) (or say that it is not present).

Other Examples:
1. Looking for a word in a dictionary
2. Debugging code
3. Rice cooking
4. Finding \( \lfloor \sqrt{\alpha} \rfloor \) of an integer \( \alpha \).

Start with a guess \( x \in [1, \alpha] \)
Check \( x^2 > \alpha \)
No. of rounds \( \log_2 \alpha \).
What if we want to output the square root as a real number?

Search space? \( a \times 2^k \)

No. of queries \( \log(a \times 2^k) = \log a + k \)

\( k \) is the no. of precision bits you are asked for.

Better than binary search?

Is there any searching scheme that can work in less than \( \log N \) queries?

Ans: No.

Argument: If the query is Yes/No type then it gives only one bit of information.

\[ \log_4 N = \frac{1}{2} \log_2 N \]

In worst case your new search space size > \( \frac{N}{2} \)

\[ \frac{1}{2}, \frac{1}{2^2}, \ldots, \frac{1}{2^{\log_2 N}} \]

Homework 1: You have two sorted arrays of integers. Assume all the entries are distinct in/around the two arrays.

Find the median of the union of two arrays by accessing only \( O(\log n) \) entries.

\( N = \text{size of arrays} \)
HW2

Given an array of integers, and a number $S$, find a pair of integers in the array whose sum is $S$.

Trivial: $\binom{n}{2} = O(n^2)$

Another application: suppose for an optimization problem you can test whether the optimal value is greater than a given number $W$.

How much time will you take to find the optimal value?

$\log(\text{Initial Range})$

What if the range is unknown? (Exponential search)

Can we find the optimal value in $O(\log(\text{optimal value}))$ queries?

Ans: query with $W = 0, 2, 2^2, 2^3, \ldots$ and stop when optimal value is $\leq 2^k$.

HW3

Let $f: \mathbb{R} \to \mathbb{R}$ be a convex function $f(x)$ is not given explicitly. You can query for $f(x)$ and $f'(x)$ at any point.

Find the point minimizing $f(x)$ (given the promise that $x^*$ exists)

Comment: $f(x) = e^x$ is convex, but has no minimizing point.
Analyzing Algorithms

- Comparing different algorithms

Running time:
Why not implement and see?

- Too many inputs
- Too many algorithms
- Processor dependent

Will count the number of basic operations.
(addition / comparison)

Asymptotic Analysis

for Input size $n$ running time $f(n)$

3$n^3$ $\rightarrow$ $O(n)$

Big O notation:
$O(n)$, $O(n^2)$, $O(n \log n)$, $O(2^n)$

$\Theta(n)$

$f(n) = \sqrt{n}$ $\in O(n)$

d$n \leq f(n) \leq c \cdot n$ $\not\in \Theta(n)$

$A \leftarrow \log n$ $\in O(n)$

$B \leftarrow n^2 + 9$ $\in O(n^2)$
Worst Case Analysis (take the worst bound over all inputs of a fixed size).

1. Why not average case analysis?

2. It's nice to have worst case guarantees and in many cases we can get it.

Describing Algorithms

Pseudocode / Textual description.

(error prone)

Implementation details.

Combination of the two
First Design Idea

Reducing to a subproblem

(Same problem on a smaller input
(Subarray, Subgraph)

Assume that you are already given a solution for the subproblem and using that try to build a solution for the original problem.

solve the subproblem using the same strategy.
Advantage: useful in analyzing the algorithm.
Implementation: recursive or iterative.

Prob 1:
Find minimum value in a given integer array.

\[ A = [a_1, a_2, a_3, \ldots, a_n] \]

subproblem: minimum among the first \( n-1 \) values.

\[ m_{n-1} \rightarrow m_n = \min (m_{n-1}, a_n) \]

<table>
<thead>
<tr>
<th>Recursive</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M(A, i) ): output min value among first ( i ) values. if ( i = 1 ) ⇒ output ( A[i] ) ( m = A[i] ) for ( i = 2 ) to ( n ) ( m = \min(m, A[i]) ) else ( \text{Output } \min(M(A, i-1), A[i]) )</td>
<td></td>
</tr>
</tbody>
</table>
Analysis:

no. of comparisons = \( n - 1 \)

Improvement?

\( n - 1 \) comparisons are necessary? \( a_i \) is min

yes, every element except the minimum one needs to be shown to be larger than something

Prob 2: Find minimum and second-minimum in an array

Naive Method: First find the minimum and then find the second-minimum.

no. of comparisons: \( n - 1 + n - 2 = 2n - 3 \)

Reducing to subproblem:

\( F_i \leftarrow \text{minimum among first } i \text{ values} \)

\( S_i \leftarrow \text{second minimum among first } i \text{ values} \)

Given \( F_{n-1}, S_{n-1} \), compute \( F_n, S_n \)

Given \( a_1, a_2, \ldots, a_n \)

If \( a_n < F_{n-1} \)

\( F_n = a_n, \quad S_n = F_{n-1} \)

Else \( F_n = F_{n-1}, \quad S_n = \min(S_{n-1}, a_n) \)

No. of comparisons = \( 2(n-2) + 1 = 2n - 3 \)
Better idea:

Subproblem on first \( n-2 \) elements.

Given \( F_{n-2}, S_{n-2} \), compute \( F_n, S_n \).

If \( a_{n+1} \leq a_n \):

1. \( F_n = a_n \)

2. \( S_n = \min(a_{n-1}, F_{n-2}) \)

else:

\( F_n = F_{n-2} \)

\( S_n = \min(a_n, S_{n-2}) \).

No. of iterations \( \frac{n}{2} \) \( \Rightarrow \) No. of comp \( \approx \frac{3n}{2} \).

There is another algorithm which has

No. of comparisons = \( n-1 + \log n \).

First, find the minimum.

Try to minimize the Candidates for second minimum.

Think about it.
Can we improve?

New Idea: randomization.

Randomly shuffle the array.
Each of \( n! \) orderings are equally likely.

\( 4, 1, 2, 3 \).

HW: how to generate a uniformly random permutation.

**Expected Running Time**

Running Time random variable.

\[ \text{probabilistic average: } \frac{1}{10} T_1 + \frac{1}{8} T_2 + \cdots \]

**Stronger**: With high prob the running time is at most \( \frac{1}{n!} \).
Finding min and second min.

\[
\text{else} \quad \text{min}_1 := A[2], \text{min}_2 := A[1]
\]

for \( i = 3 \) to \( n \)

\[
\text{if } (A[i] \leq \text{min}_1) \\
\quad \text{min}_2 := \text{min}_1, \quad \text{min}_1 := A[i]
\]

\[
\text{else} \\
\quad \text{if } (A[i] < \text{min}_2) \\
\quad \quad \text{min}_2 := \text{A}[i]
\]

\[ \text{else} \quad \text{do nothing.} \]

Depending on the input order, some iterations see 2 comparisons and some see only 1.

Worst case input order

minimum is at first position.

Best case input order

Sorted in decreasing order.
Hope is that by random shuffling a bad input order becomes a good input order. (with large probability)

Let's see more closely.

In $i$-th iteration

Good event $A[i] \leq \text{current min}$ $[P = \frac{1}{i}]$

Bad event $A[i] > \text{current min}$

Which one seems more likely?

Change the algorithm so that 2 comparisons become less likely.

```
for $i = 3$ to $n$
    if ($A[i] > \text{min}_2$)
        Do nothing.
    else if ($A[i] > \text{min}_1$)
        \text{min}_2 := A[i]
    else
        \text{min}_1 := A[i]
```
min no. of total comparisons $\eta - 1$
max no. of total comparisons $2\eta - 3$

Want to compute expected no. of total comparisons

Definition Expectation of random variable $X$.

$$E[X] = \sum_{x} x \cdot Pr [X = x]$$

Let $X$ be the total no. of comparisons.

$$E[X] = \sum_{x=\eta-1}^{2\eta} x \cdot Pr \left[ \text{exactly } x \text{ comparisons in total} \right]$$

Looks difficult to estimate $\eta!$.

$$= \sum_{\sigma} \frac{1}{\eta!} \times \text{no. of comparisons for order } \sigma$$

Claim:

Expected no. of total comparisons

$$= 1 + \sum_{i=3}^{\eta} \text{Expected no. of comparisons in iteration } i$$
Obvious: no of total comparisons

\[ = 1 + \sum_{i=3}^{n} \text{no of comparisons in iteration } i \]

\[ \text{In linearity of expectation} \]

\[ X = \sum_{i} x_{i} \]

\[ E[X] = \sum_{i} E[x_{i}] \]

\[ E[x_{i}] = \sum_{o} \frac{1}{n!} \times (\text{no of comparisons in iteration } i \text{ on order } o) \]

\[ = 1 \cdot \Pr(x_{i}=1) + 2 \cdot \Pr(x_{i}=2) \]

\[ \Pr[X_{i}=2] \]

\[ = \Pr\left[ A[i] \leq \text{second-min } A[1 \ldots i-1] \right] \]

\[ = \Pr\left[ A[i] \text{ is min or second-min among } A[1 \ldots i] \right] \]

\[ = \Pr\left[ \text{min or second-min among the first } i \text{ falls at the } i\text{-th position} \right] \]

\[ = \frac{2}{i}. \]
Reasoning: each element among the first $i$ is equally likely to fall at the $i$-th position.

Thus, the probability that min among first $i$ will fall at the $i$-th position is $1/i$.

Similarly probability for the second-min is $1/i$.

Adding up the two, we get $2/i$.

Alternate proof for $p(2/i)$.

We know that each permutation is equally likely.

So, we need to find the number of permutations s.t. $A[i]$ is min or second-min among $A[1..i]$, and divide by $n!$.

1. First lets count the no. of possibilities for $A[i+1, i+2, \ldots, n]$.

\[
\text{no. of ways to choose } A[i+1] \quad \text{from remaining elements} \times 
\text{then no. of ways to choose } A[i+2] \quad \text{from remaining elements} \times 
\vdots \times 
\text{then no. of ways to choose } A[n] \quad \text{from remaining elements}
\]

\[= n \times (n-1) \times (n-2) \times \ldots \times (n-i)\]
Once we have fixed $A[i+1, ..., n]$ out of the remaining elements, min or second-min can be put in $A[i]$. That means 2 ways to choose.

Finally for remaining $i-1$ elements, there are $(i-1)!$ ways of arranging them in $A[1..i-1]$.

Overall no. of permutations

$$= n(n-1) ... (n-i) \times 2 \times (i-1)!$$

After dividing by $n!$, we get $2/i$.

Randomized min & second min continued Aug 9

$$E[x_i] = 1 \cdot Pr(x_i=1) + 2 \cdot Pr(x_i=2)$$

$$= 1 \cdot (1 - 2/i) + 2 \cdot 2/i$$

$$= 1 + 2/i$$

Expected no. of total comparisons

$$E[x] = 1 + \sum_{i=3}^{n} E[x_i]$$

$$= 1 + \sum_{i=3}^{n} \left(1 + \frac{2}{i}\right) = n-1 + \sum_{i=3}^{n} \frac{2}{i}$$
\[
\frac{f(x)}{x} = \frac{1}{x}
\]

\[
\text{Expected no. of comparisons} \leq n + 2\log n
\]

Que: 1) Can we show that the no. of comparisons is around the expectation with a good prob?

\[
\Pr [ X > n + 2\delta \log n ] \leq \frac{1}{\delta}
\]

Much better concentration guarantees can be shown via advanced methods.

HW: Implement and see \( f(n) - n \) vs. \( n \)

2) Min. Second-min. third min

3) Randomized Quicksort \( \text{[non-trivial]} \)

4) Expected height of a randomly constructed binary search tree.
\( \text{[don't know how hard]} \).
Maximum Subarray Sum problem.

Subarray - contiguous subset.

Given an integer array (possibly with negative entries), find the subarray with maximum sum.

Naive Algorithm:

Go over all possible subarrays and find the one with maximum sum.

\[ \text{for } \text{start} = 1 \text{ to } n \]

\[ S = 0 \]

\[ \text{for } \text{end} = \text{start} \text{ to } n \]

\[ S = S + A[\text{end}] \]

\[ \text{curr-max} := \text{Max}(\text{curr-max}, S) \]

\[ \text{for } (i = \text{start} \text{ to } \text{end}) \]

\[ n + n-1 + n-2 \]

\[ O(n^3) \]

New running time \( = O(n^2) \).

Can we improve?
Think about how the subproblem idea can be applied here.

Assume $\text{MaxSubarray}(n-1)$ is given.

Can we compute $\text{MaxSubarray}(n)$ using it?

- Subarrays of $A$ are of two kinds.
  - Those including $A[n]$
  - Those not including $A[n]$

$$\text{MaxSubarray}(n) = \max \left\{ \begin{array}{l}
\text{MaxSubarray}(n-1) \\
A[n] \\
\sum[n-1..n] \\
\sum[n-2..n] \\
\vdots \\
\sum[1..n] \\
0(n) \end{array} \right\}$$

$$T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$$
Improvements?

Ask the subproblem to solve more:

\[
\max (\text{sum}[1 \cdots n], \text{sum}[n-2 \cdots n], \ldots, \text{sum}[1 \cdots n]) = \max \left( A[n] + \text{sum}[n-1], A[n] + \text{sum}[n-2 \cdots n-1], \ldots, A[n] + \text{sum}[1 \cdots n-1] \right)
\]

\[
= A[n] + \max (\text{sum}[n-1], \text{sum}[n-2 \cdots n-1], \ldots, \text{sum}[1 \cdots n-1])
\]

Subproblem: \( \max \text{Subarray}(n-1) \) & \( \max \text{Suffix}(n-1) \)

\[
\max \text{Subarray}(n) = \max \left\{ \begin{array}{l}
\max \text{Subarray}(n-1) \\
\max \text{Suffix}(n-1) + A[n] \\
A[n]
\end{array} \right. 
\]

\[
\max \text{Suffix}(n) = \max (A[n], \max \text{Suffix}(n-1) + A[n])
\]

\[
\max \text{Suffix} = A[i] \quad \max \text{Subarray} = \max (0, A[i])
\]

for \( i = 2 \) to \( n \)

\[
\max \text{Subarray} = \max \left\{ \begin{array}{l}
\max \text{Subarray} \\
\max \text{Suffix} + A[i] \\
A[i]
\end{array} \right. 
\]

\[
\max \text{Suffix} = \max \left\{ A[i], \max \text{Suffix} + A[i] \right\}
\]
\[ T(n) = O(n) \]

**Principle Used:**

When designing recursive / inductive idea, sometimes it is useful to solve a more general or harder problem.

**HW** Longest Increasing Subsequence

2 4 3 9 5 10

**Exponentiation:**

Given \( a, n \) compute \( a^n \).

\[
\text{Exp}(a, n) = \begin{cases} 
\text{Exp}(a, n-1) \times a & \text{if } n \text{ is even} \\
(n-1) & \text{if } n \text{ is odd}
\end{cases}
\]

- **Multiplication unit cost:**
- No. of multiplication = \( n - 1 \)

- if \( n \) is even
  \[ a^n = (a^{n/2})^2 \]

- if \( n \) is odd
  \[ a^n = a \cdot \left[ a^{\frac{n-1}{2}} \right]^2 \]
\[ T(n) = 2 + T\left(\frac{n}{2}\right) \]
\[ T(n) \leq 2 \log_2 n. \]

\[ a^7 = (a^3)^2 \times a \quad 2 \text{ mult.} \]
\[ a^3 = (a)^2 \times a \quad 2 \text{ mult.} \]

\[ 4 \text{ mult.} \]

\[ \begin{align*}
\{ a^{15} &= (a^7)^2 \times a \quad 2 \text{ mult.} \\
\{ a^7 &= (a^3)^2 \times a \quad 2 \text{ mult.} \\
\{ a^3 &= (a)^2 \times a \quad 2 \text{ mult.} \}
\end{align*} \]

\[ 2(\log n - 1) \]

\[ \begin{align*}
\{ a^2 &= a \times a \quad 1 \text{ mult.} \\
\{ a^5 &= (a^2)^2 \times a \quad 2 \text{ mult.} \\
\{ a^{15} &= (a^5)^2 \times a^5 \quad 2 \text{ mult} \}
\end{align*} \]

5 mult.

Given \( n \), what is the smallest number of multiplications needed to compute \( a^n \)?

Can you design an efficient algorithm to find the smallest number of multiplications required?

Matrix Exponentiation.

\[ F_n = F_{n-1} + F_{n-2} \]

Can you compute the \( n^{\text{th}} \) Fibonacci number in \( O(\log n) \) operations? H/W
Divide and Conquer

- Divide the problem into multiple subproblems of size \( \eta/2 \)
- Combine the solutions of the subproblems and build a solution for the original problem.

Example: Mergesort.

\[
T(\eta) = a \cdot T(\eta/2) + f(\eta)
\]

\( T(\eta) \) is the running time
\( n \) is the number of subproblems.

Divide and Conquer might improve the running time for example:

\( O(\eta^2) \rightarrow O(\eta \log \eta) \)

This technique doesn’t usually help with designing polynomial time algorithms when none is known.

Hidden Surface Removal

Given \( n \) lines (non-vertical, infinite), we need to find out which ones are visible from \( y = \infty \).

Example:

Visible \( 1 \ 2 \ 3 \ 6 \)
Hidden \( 4 \ 5 \)
Input: \( y = m_1 x + c_1, \ y = m_2 x + c_2, \ldots, \ y = m_n x + c_n \)

Formally at \( x = x_0 \), that line is visible which maximizes \( m_i x_0 + c_i \).

We need to find lines that are visible at some \( x \) for each value of \( x \).

Naive solution: find the line giving \( \max m_i x + c_i \).

Infinitely many \( x \)?

No. Find intersection points for each pair of lines. These are the only interesting \( x \) values.

\[
\text{Time} = O(n^2 \log n + n^3) = O(n^3)
\]

Can we do better?

Given a solution for \( n-1 \) lines, can we compute a solution for \( n \) lines.

What should the solution for \( n-1 \) lines look like?
form: a list of $x$-values, $x_1, x_2 \ldots x_k$ and the line segment visible in the interval $[x_i, x_{i+1}]$ for each $i$.

To insert the new line
We can check the $y$ value of the new line at each critical $x$ value.
Whether above the existing segment or below it?

$T(n) = T(n-1) + O(n)$

$T(n) = O(n^2)$

Better Idea?

**Observation:** The new line intersects the existing set of visible segments at at most two points.

**Why:**
1. As we go from left to right, slopes of visible lines are increasing.
   
   $\text{slope } (x_i, x_{i+1}) < \text{slope } (x_{i+1}, x_{i+2})$

2. Let $m_n$ be the slope of the new line $l_n$
   $l_n$ cannot intersect with more than one line segment with slope greater than $m_n$
Similarly, in cannot intersect with more than one line segment with slope smaller than \( m_n \).

Can we do binary search for the two intersection points?

Divide the current set of segments into two groups: slope \( > m_n \) and slope \( < m_n \).

Do a binary search for the intersection point in each group.

\( O(n) \) to update

How much time required update our solution?

is there a data structure which allows insertion and deletion in \( O(\log n) \) time?

Can we also do search in that data structure?

HW: Convince yourself that with a balanced binary search tree we can do insertions and deletions in \( O(\log n) \) time.

Moreover, search for the two segments intersecting the new line can also be done in \( O(\log n) \) time.

\( \Rightarrow \) Total running time = \( O(n \log n) \).
New Algorithm

If we sort the lines beforehand w.r.t. their slopes, we don't need to worry about a sophisticated data structure.

- $O(\log n)$
- $O(1)$

binary search

suffix

The new line has a slope greater than the current segments. Has exactly one intersecting segment. Can be found with binary search.

Update is simpler now:

- Have to delete a set of hidden segments from the end.
- And insert the new segment at the end.

Total running time = $O(n \log n)$

Conclusion: sometimes order of the input is important.

Divide and conquer approach. $O(n \log n)$

Divide the set of lines into two halves (arbitrary).

Given the solutions for each set, can we "merge" them to build a solution for the whole set?
Sol 1 (red)
\[ x_1, x_2, x_3, x_4, \ldots \]
\[ y_1, y_2, y_3, y_4, \ldots \]

Sol 2 (blue)
\[ p_1, p_2, p_3, p_4, \ldots \]
\[ q_1, q_2, q_3, q_4, \ldots \]

"Merging"
traversing the x co-ordinates from left to right.

Two pointers for the two lists.

Say, current pointers are at \( x_i \) & \( p_j \)

Consider the min of \( x_i \) & \( p_j \), say \( x_i \)

Compare the y values of both the curves at \( x_i \) and see whether the red curve is above or the blue curve.

If the order of red and blue curves was different at the previous x-coordinate, then need to introduce an intersection point.
Say $x_i, y_i$ is a point in the red list. If the $y$-value in the blue curve at $x_i$ is smaller than $y_i$, then insert $(x_i, y_i)$. Else, don’t insert anything.

HW: Write a code for the "merge" algorithm.

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

$$T(n) = O(n \log n).$$

Three algorithms:

1. Process lines one by one.
   Need a data structure like balanced binary search tree.

2. First sort the lines w.r.t. their slopes.
   No need of a sophisticated data structure.
   But need the complete input set of lines beforehand.

3. Divide and conquer.

   All three algorithms take $O(n \log n)$ time.
Integer Multiplication

Bit complexity

Adding two \( n \)-bit numbers \( \rightarrow \mathcal{O}(n) \)

Multiplying two \( n \)-bit numbers

\[ a \times b \rightarrow \text{add } a, \ b \text{ times} \]

School method

\[
\begin{array}{c}
\text{101} \\
\text{000} \\
\text{1010} \\
\text{1111}
\end{array}
\]

\[ 0(n^2) \]

Can we do better?

Karatsuba [1960] \( 0(n^{1.58}) \)

Let's first talk about squaring.

\( a \leftarrow n \text{ bit integer. Find } a^2 \)

Let's try to reduce it to squaring of an \( n-1 \) bit integer

\( a = 2a' + e \) \( \text{2 left shifts} \)

\( a^2 = (2a' + e)^2 = 4a'^2 + e^2 + 4a'e \) \( \text{n+2 bits} \)

\( \mathcal{T}(n) = 0(n^2) \) \( \Rightarrow \mathcal{T}(n) = \mathcal{T}(n-1) + \mathcal{O}(n) \)

\( \text{2 left shifts} \)
How about divide and conquer?
Reducing to squaring \( \frac{n}{2} \) bit integers.

\[
\begin{align*}
A &= a_i \cdot 2^{\frac{n}{2}} + a_0 \\
A^2 &= (2^{\frac{n}{2}} a_i + a_0)^2 \\
&= 2^n a_i^2 + a_0^2 + 2 \cdot 2^{\frac{n}{2}} a_i a_0
\end{align*}
\]

left shift by \( n \) bits

\[
T(\frac{n}{2}) \quad T(\frac{n}{2})
\]

Can we compute \( a_i a_0 \) via squaring?

\[
2 \cdot a_i a_0 = (a_i + a_0)^2 - a_i^2 - a_0^2
\]

\[
a_i^2 = 2^n a_i^2 + a_0^2 + 2^{\frac{n}{2}} ((a_i + a_0)^2 - a_i^2 - a_0^2)
\]

\[
T(n) = 3 \cdot T(\frac{n}{2}) + O(n)
\]

\[
T(n) = O(n \log_2 3) \approx O(n^{1.58})
\]
What about multiplication?

Multiplication directly (without going via squaring)

\[ a \times b \]

\[ a = a_1 \cdot 2^{n/2} + a_0 \]

\[ b = b_1 \cdot 2^{n/2} + b_0 \]

\[ ab = a_1 b_1 \cdot 2^n + a_0 b_1 \cdot 2^{n/2} + a_1 b_0 \cdot 2^{n/2} + a_0 b_0 \]

can these four terms \( a_1 b_1, a_0 b_1, a_1 b_0, a_0 b_0 \) be computed somehow with three multiplications?

Hint: first compute \((a_1 + a_0) (b_1 + b_0)\)

\( n/2 \) bits

two more multiplications allowed.
Can we instead divide into three parts?

Squaring \( a = a_2 \cdot 2^{2n/3} + a_1 \cdot 2^{n/3} + a_0 \)

\[
a^2 = a_2^2 \cdot 2^{4n/3} + 2a_2a_1 \cdot 2^{2n/3} + (a_1^2 + 2a_0a_2) \cdot 2^{2n/3} + 2a_1a_0 \cdot 2^{n/3} + a_0^2
\]

\[T(n) = \alpha T(n/3) + O(n)\]

\[T(n) = O(n^{\log_3 \alpha})\]

\[\log_3 6 = 1.63\]

\[\log_3 5 = 1.46\]

Can we compute the desired terms with 5 squarings?