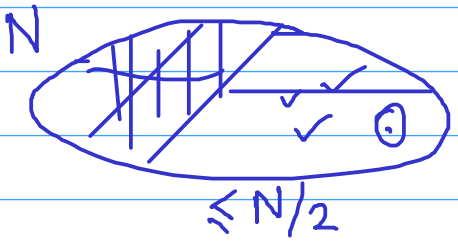
Lecture 1 CS601 July 29 Thursday

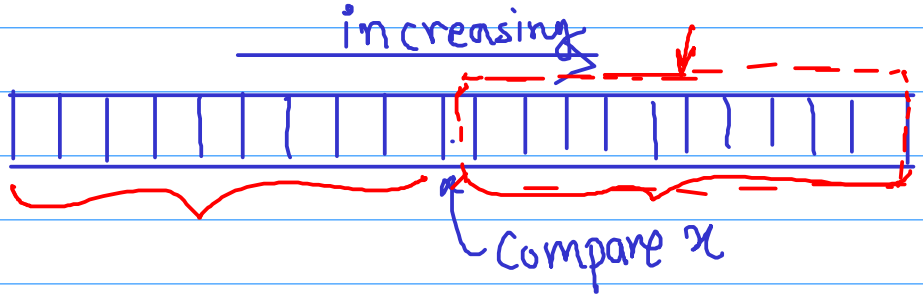
Binary Search (and variants)

Applicable when with one query, the search space size can be reduced by half.
( )



No. of queries log N

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



Other Examples:

① Looking for a word in a dictionary

② Debugging Code

③ Rice cooking

① Finding [Square root] of an integer a.

Start with a guess $x \in [1, a]$

Check $x^2 > a$

No. of rounds $\log a$.

What if we want to output the square root as a real number?

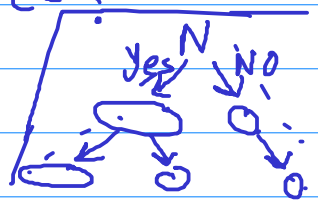
Search space? $a \times 2^k$

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

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

Better than binary search?

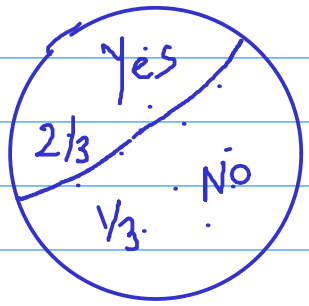
Is there any searching scheme that can work in less than $\log_2 N$ queries?



Ans: NO.

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

$\log_4 N = \frac{1}{2}\log_e N$



In worst case your new search space size $\geq N/2$.

$1/2 \cdot 1/2 \cdots 1/2^{\log_2 N}$

Homework ① You have two sorted arrays of Integers Assume all the entries are distinct in/across the two arrays.

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

n = 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 (initial Range)

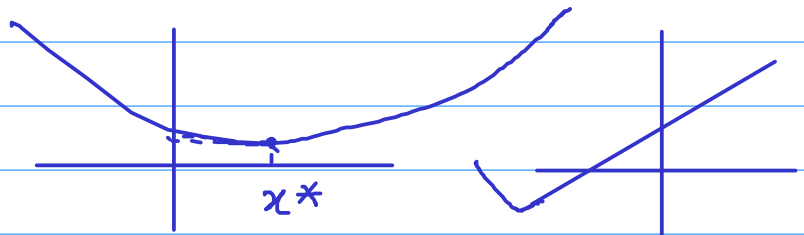
What if the range is unknown? (Exponential Search)

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



[equivalently $f'(x)$ is non-decreasing]

$f(x)$ 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 $f(n)$

$3n-5$ $2n+3$, $5n^2-n+4$.

$\rightarrow O(n)$

Big-O notation.

$O(n)$, $O(n^2)$, $O(n \log n)$, $O(2^n)$

$\Theta(n)$

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

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

A $\leftarrow$ $100 \cdot n$ $\leftarrow O(n)$

B $\leftarrow$ $n^2 + 9$ $\not\leftarrow O(n^2)$

Worst Case Analysis { take the worst bound over all inputs of a fixed size

Why?

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

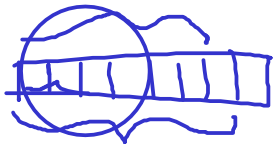
Pseudocode / Textual description.

(error prone)

implementation details.

Combination of the two

First Design Idea



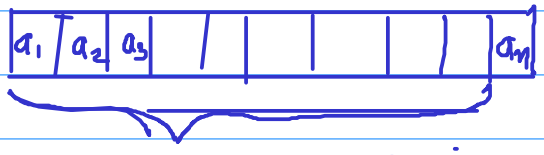
Reducing to a subproblem

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 iteger array.

$A =$ 

subproblem: minimum among the first $n-1$ values.

m_{n-1}

$\rightarrow \quad m_n = \min(m_{n-1}, a_n)$

Recursive	Iterative
· M (A, i): output min value among first i values. if i=1 ⇒ output A[1] else Output min (M(A,i-1), A[i])	m = A[1] for i = 2 to n m = min(m, A[i])

Aug 3

Analysis:

no. of comparisons = $n-1$

$a_j \leftarrow a_k \leftarrow a_i$

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.

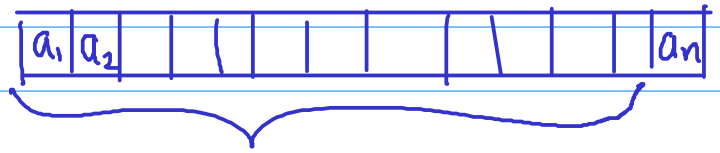
no. of comparisons: $n-1 + n-2 = \underline{2n-3}$

Reducing to subproblem:

$F_i \leftarrow$ minimum among first i values

$S_i \leftarrow$ second minimum among first i values.

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



$a_n < F_{n-1} < S_{n-1}$

$F_{n-1} \leq S_{n-1}$

$F_{n-1} \leq a_n$

- If $a_n \leq F_{n-1}$

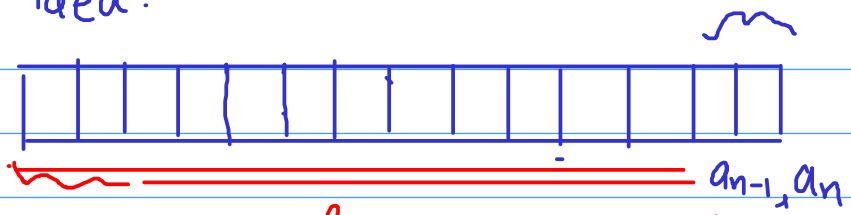
  $F_n = a_n$, $S_n = F_{n-1}$

Else

$F_n = F_{n-1}$, $S_n = \underline{\min(S_{n-1}, a_n)}$.

No. of comparisons = $2(n-2)+1 = \underline{2n-3}$.

Better Idea:



Subproblem on first $n-2$ elements.

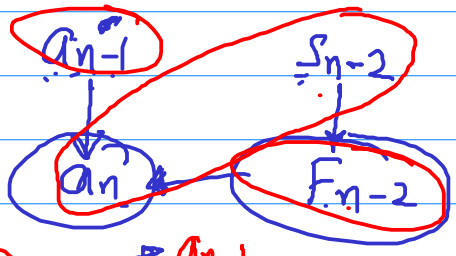
Given F_{n-2}, S_{n-2}, Compute F_n, S_n.

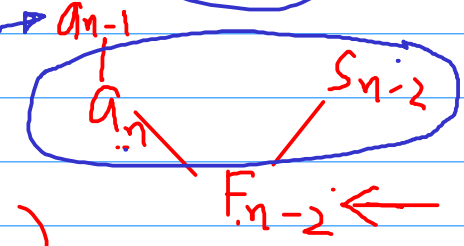
If $a_{n-1} > a_n$ ①

If $a_n < F_{n-2}$ ← ②

$F_n = a_n$

$S_n = \min(a_{n-1}, F_{n-2})$ ③





else

$F_n = F_{n-2}$

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

else

No. of iterations $\frac{n}{2}$ ⟹ 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.

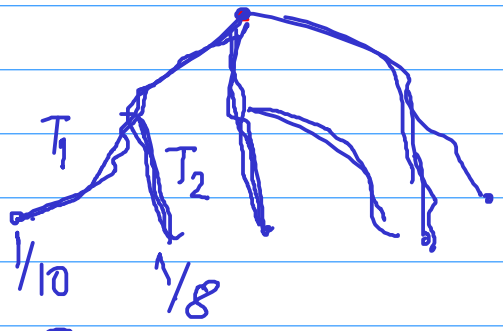
Randomly shuffle the array.
Each of n! orderings are equally likely.

4, 1, 2, 3. HW: how to a generate a uniformly random permutation.

Expected Running Time

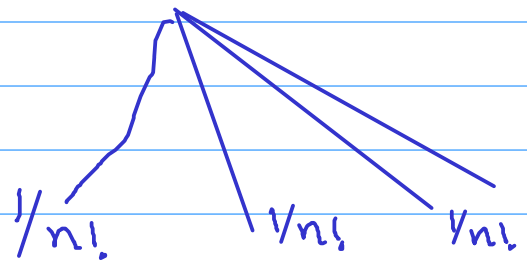
Running Time random variable.

→ probabilistic average.



$$\left[\frac{1}{10} \times T_1 + \frac{1}{8} \times T_2 + \cdots\right.$$

Stronger: With high prob the running time is at most ≤.



Aug 5

Finding min and second min.

```
if (A[1] < A[2])    min1 := A[1], min2 := A[2]
else                min1 := A[2], min2 := A[1]

for i = 3 to n
    if (A[i] < min1)
        min2 := min1
        min1 := A[i]
    else
        if (A[i] < min2)
            min2 := A[i]
        else
            do nothing.
```

[10, 20, 30, 40 ....]

A[i] > min1

[10, 30, 25, 20, 40]

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

Worst case input order
→ minimum is at first position.

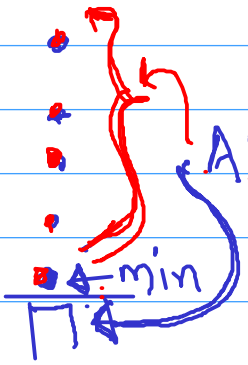
Best case input order

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

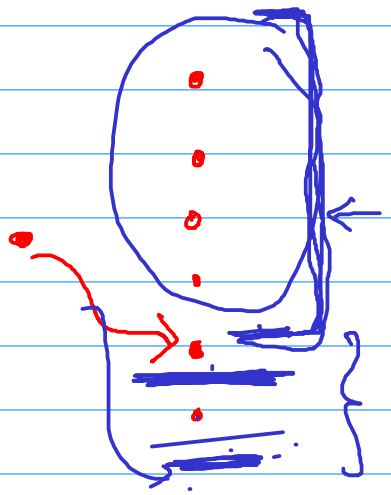


Good event $A[i] \leq$ currentmin $\left[Pr = \frac{1}{i}\right]$

Bad event $A[i] >$ 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] > min2)
        Do nothing.
    else
        if (A[i] > min1)
            min2 := A[i]
        else
            min2 := min1
            min1 := A[i]
```

Que. What if A[2] is the largest. ~~Worst Case~~

(50, 40, 30, 20, 10)

min no. of total comparisons $n-1$

max no. of total comparisons $2n-3$

Want to compute expected no. of total comparisons

Definition Expectation of random variable X.

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

Let X be the total no. of comparisons.

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

1, 2, 5, 4, 3

Looks difficult to estimate

$$\Pr[\text{exactly } x \text{ comparisons}] = \frac{\text{no. of input orders with exactly } x \text{ compari}}{n!}$$

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

Claim:

Expected no. of total comparisons

$$= 1 + \sum_{i=3}^{n} \text{expected no. of comparisons in iteration } i$$

Obvious: no. of total comparisons

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

Thm Linearity of expectation

$$X = \sum_i X_i$$

$$E[X] = \sum_i E[X_i]$$

$$E[X_i] = \sum \frac{1}{n!} \times \left(\begin{array}{c}\text{no. of comparisons} \\ \text{in iteration } i \text{ on} \\ \text{order}\end{array}\right)$$

$$= 1 \cdot Pr(X_i = 1) + 2 \cdot Pr(X_i = 2)$$

over all possible

$Pr[X_i = 2]$?

$$= Pr[A[i] < \text{second-min}(A[1 \ldots i-1])]$$

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

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

$$= 2/i.$$

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 prob $2/i$.

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 $A[i+1, i+2, \ldots n]$.

$$= n \times (n-1) \times (n-2) \times \cdots (n-i)$$

- n: no. of ways to choose $A[i+1]$
- $(n-1)$: then no. of ways to choose $A[i+2]$ from remaining elements
- $(n-2)$: then no. of ways to choose $A[i+3]$ from remaining elements
- $(n-i)$: then no. of ways to choose $A[n]$ from remaining elements

② Once we have fixed $A[i+1, \ldots, n]$ out of the remaining elements, min or second-min can be put in $A[i]$

That means 2 ways to choose.

③ Finally for remaining $i-1$ elements, there are $(i-1)!$ ways of arranging them in $A[1..i-1]$

Overall no. of permutations

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

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

Randomized min & second min continued Aug 9

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

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

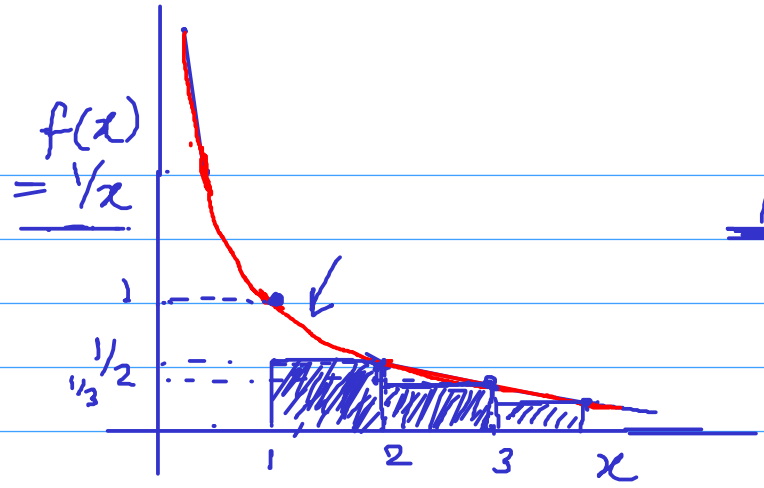
$$= 1 + 2/i$$

Expected no. of total comparisons

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

$$= 1 + \sum_{i=3}^{n} (1 + \tfrac{2}{i}) = n - 1 + \sum_{i=3}^{n} \tfrac{2}{i} \approx 2 \log n$$

$E[X] \leq n + 2\log_e n$



HW $\sum_{i=2}^{n} \frac{1}{i} \leq \log_e n$

$\int 1/x \, dx$

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?

HW $Pr[X > n + 2\delta \log n] \leq 1/\delta$

Much better concentration guarantees can be shown via advanced methods.

HW Implement and see. $f(n) - n$ vs. n

---

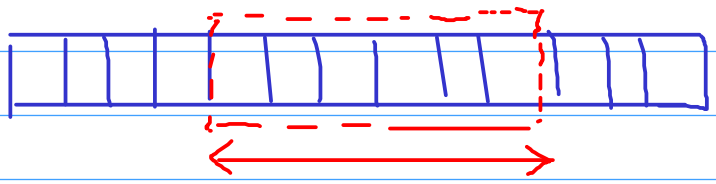
② Min, Second-min, third min

③ Randomized Quicksort [non-trivial]

④ Expected height of a randomly constructed binary search tree.

[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

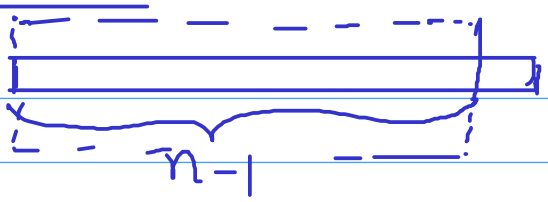
```
curr-max;
for start = 1 to n.
        s = 0
    for end = start to n
        s = s + A[end];
        curr_max := Max(curr-max, s)
    for (i = start to end)
```

$n + n-1 + n-2$ $\quad O(n^3)$

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

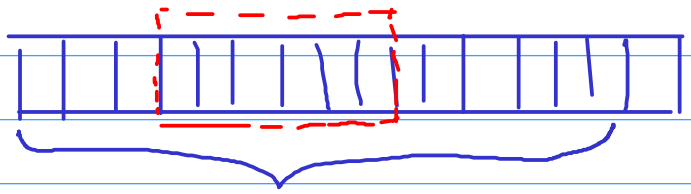
Can we improve?

Subproblem



Think about how the subproblem idea can be applied here.

Aug 10

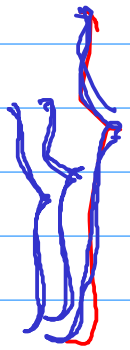


Assume MaxSubarray (n-1) is given.

Can we compute MaxSubarray (n) using it ?

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

MaxSubarray (n) = Max { MaxSubarray (n-1) ✓, A[n], Sum [n-1 .. n], Sum [n-2 .. n], ⋮, Sum [1 .. n] }



O(n)

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

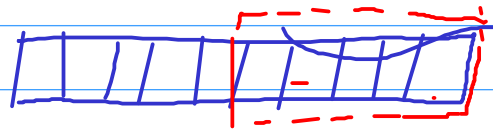
Improvements?

Ask the subproblem to solve more.

$$\max(sum[n-1..n],\ sum[n-2..n],\ \ldots,\ sum[1..n])$$

$$= \max(A[n]+Sum[n-1],\ A[n]+sum[n-2\cdots n-1],\ \ldots\ A[n]+sum[1..n-1])$$

$$= A[n] + \max(sum[n-1],\ sum[n-2\cdots n-1],\ \ldots,\ sum[1..n-1])$$



Subproblem: maxSubarray(n-1) & maxSuffix(n-1)

$$maxSubarray(n) = \max \begin{cases} maxSubarray(n-1) \\ maxSuffix(n-1)+A[n] \\ A[n] \end{cases}$$

$$maxSuffix(n) = \max(A[n],\ maxSuffix(n-1)+A[n])$$

maxSuffix = A[1] maxSubarray = max(0, A[1])

for (i = 2 to n)

$$maxSubarray = \max \begin{cases} maxSubarray \\ maxSuffix + A[i] \\ A[i] \end{cases}$$

$$maxSuffix = \max \begin{cases} A[i] \\ maxSuffix + A[i] \end{cases}$$

$T(n) = O(n)$

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 1 9 5 10

Exponentiation.

Given a, n compute a^n.

Multiplication unit cost.

Exp(a, n)

Exp(a, n) = Exp(a, n-1) x a

No. of multiplication = $n-1$

if n is even

$$a^n = (a^{n/2})^2$$

if n is odd

$$a^n = a \cdot \left[a^{\frac{n-1}{2}}\right]^2$$

$$T(n) = 2 + T\left(\frac{n}{2}\right)$$

$$T(n) \leq 2 \log_2 n.$$

$a^7 = (a^3)^2 \times a$ — 2 mult.

$a^3 = (a)^2 \times a$ — 2 mult.

4 mult.

$a^{15} = (a^7)^2 \times a$ — 2 mult.

$a^7 = (a^3)^2 \times a$ — 2 mult.

$a^3 = (a)^2 \times a$ — 2 mult.

$2(\log n - 1)$

6 mult.

$a^2 = a \times a$ — 1 mult.

$a^5 = (a^2)^2 \times a$ — 2 mult.

$a^{15} = (a^5)^2 \times a^5$ — 2 mult

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.

[Pingala, 2nd-3rd Century BC]

[LSLSS ... k

2^k.

$$F_n = F_{n-1} + F_{n-2}$$

Can you compute the n^{th} Fibonacci number in $O(\log n)$ operations? HW

Aug 12

Divide and Conquer

- Divide the problem into multiple subproblems of size $n/2$

- Combine the solutions of the subproblems and build a solution for the original problem.

Example Mergesort.

$$T(n) = a \cdot T(n/2) + f(n)$$

(a: no. of subproblems; $f(n)$: merge)

Divide and conquer might improve the running time for example

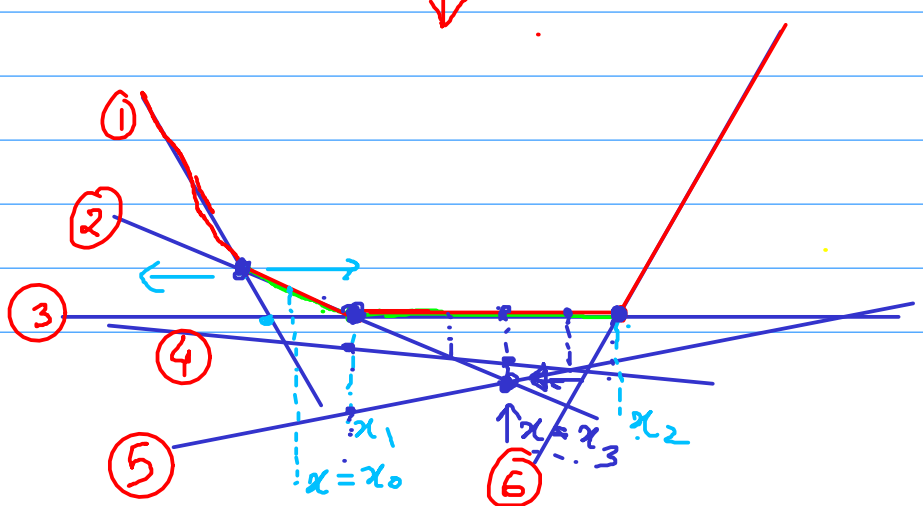
$$O(n^2) \longrightarrow O(n \log n)$$

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

Hidden Surface Removal

Give 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 : $y = m_1 x + c_1$, $y = m_2 x + c_2$, ..., $y = m_n x + c_n$

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.

Naive solution: for each value of x, find the line giving max $m_i x + c_i$

Infinitely many x?

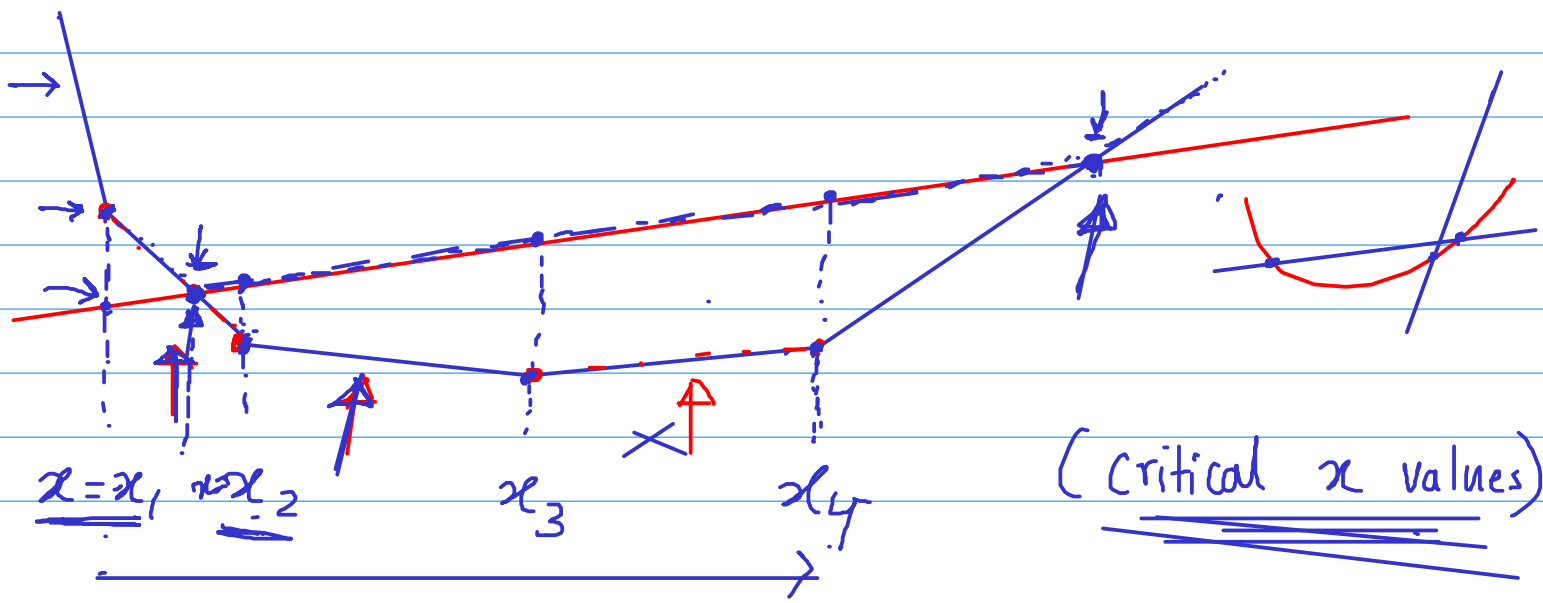
No. Find intersection points for each pair of lines. These are the only interesting x values.

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?



solution form: a list of x-values, $x_1, x_2 \dots x_k$ 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?

$T(n) = T(n-1) + O(n)$ $\quad T(n) = O(n^2)$

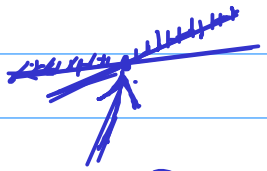
Better Idea?

Aug 16

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

Why:

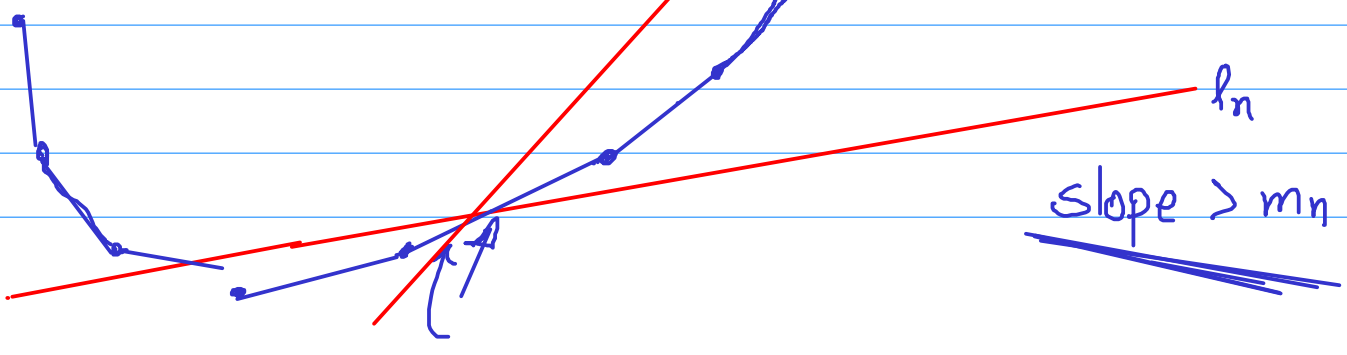
① As we go from left to right, slopes of visible lines are increasing.



slope $(x_i, x_{i+1}) <$ slope (x_{i+1}, x_{i+2})

② Let m_n be the slope of the new line l_n

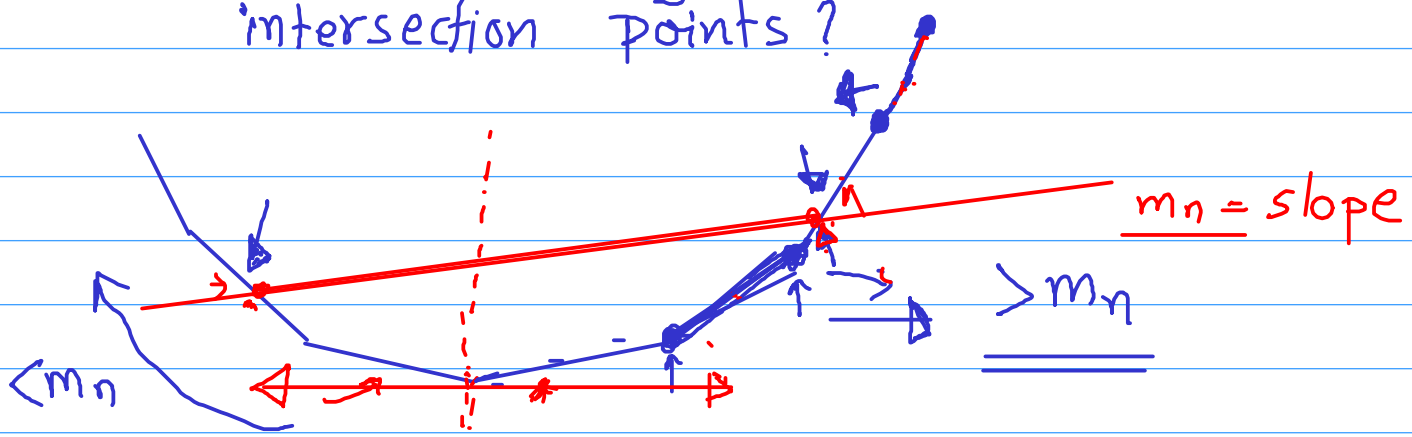
l_n cannot intersect with more than one line segments with slope greater than m_n



Similarly,

l_n cannot intersect with more than one line segments 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?

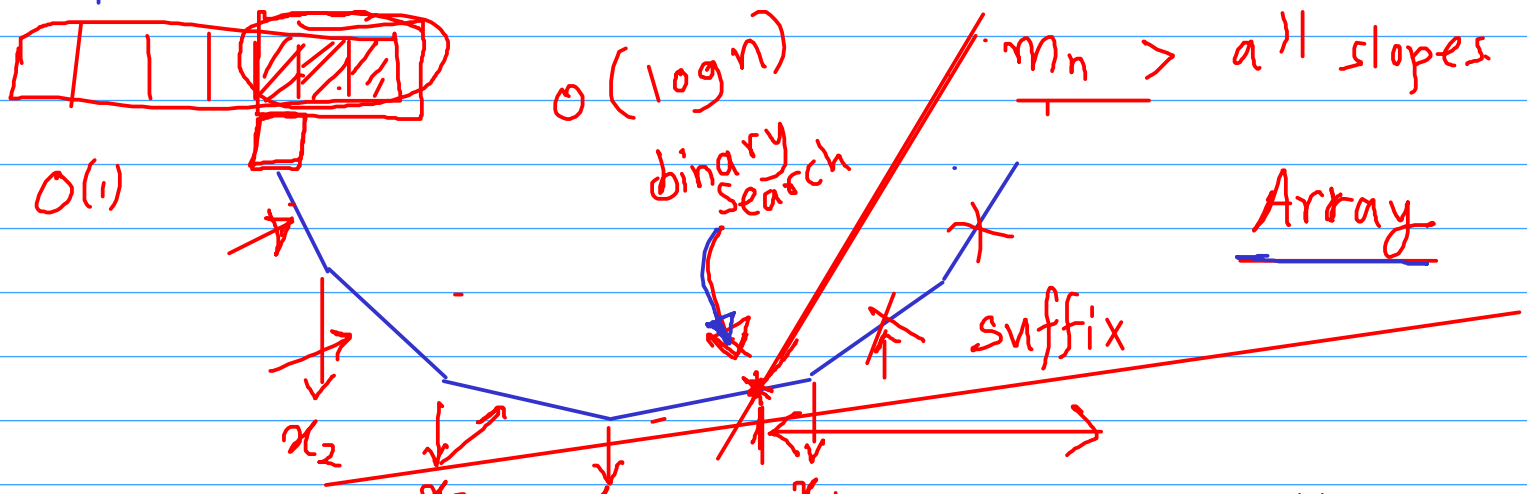
Can we also do search in that data structure?

H/W: 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 the we don't need to worry about a sophisticated data structure.



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 = $O(n \log n)$

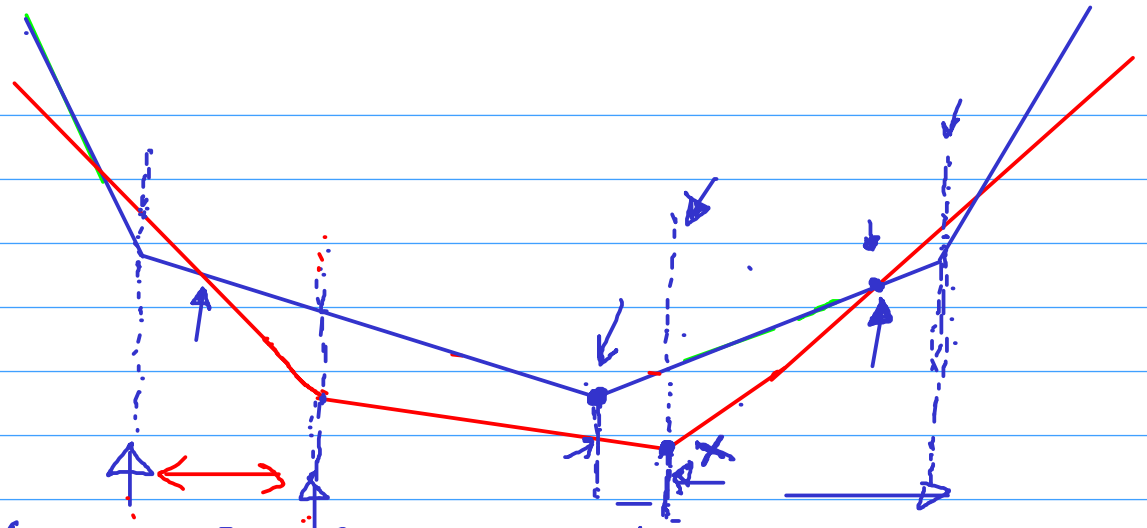
Conclusion: sometimes order of the input is important.

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?

Aug 17



Sol 1 (red) x_1 x_2 x_3 x_4 ...

y_1 y_2 y_3 y_4

sol 2 P_1 P_2 P_3 P_4 ....

(blue) q_1 q_2 q_3 q_4 .....

"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 $\longrightarrow O(n)$

Multiplying two n-bit numbers

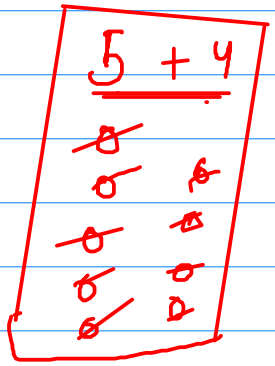
$a \times b \rightarrow$ add a, b times ($b \uparrow 2^n$)

school method

```
      101
      110
    -----
      000   <- n bits
     1010   <- n+1
    10100   <- 2n-1 bits
    -----
    11110
```

(n rows)

$O(n^2)$



Can we do better?

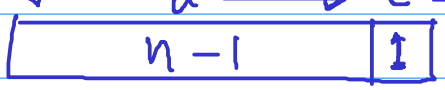
Karatsuba [1960] $O(n^{1.58})$

Let's first talk about squaring.

$a \leftarrow$ n bit integer. Find a^2.

let's try to reduce it to squaring of an $n-1$ bit integer

$a = 2a' + \varepsilon$ ($\varepsilon = 0$ or 1)



$$a^2 = (2a'+\varepsilon)^2 = \underbrace{4 \cdot a'^2}_{\sim 2n+3 \text{ bits}} + \varepsilon^2 + \underbrace{4a'\varepsilon}_{n+2 \text{ bits}}$$

2 left shifts

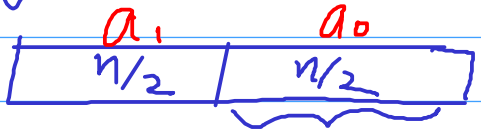
2 left shifts

$T(n) = O(n^2) \Leftarrow T(n) = T(n-1) + O(n)$

How about divide and conquer?

Reducing to squaring $n/2$ bit integers.

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



$a^2 = (2^{n/2} a_1 + a_0)^2$

$= 2^n \cdot a_1^2 + a_0^2 + 2 \cdot 2^{n/2} \cdot a_1 \cdot a_0$

left shift by n bits

a_1^2 → $T(n/2)$

a_0^2 → $T(n/2)$

$a_1 \cdot a_0$ → ?

Can we compute $a_1 \cdot a_0$ via squaring?

→ $2 \cdot a_1 \cdot a_0 = (a_1 + a_0)^2 - a_1^2 - a_0^2$

$a^2 = 2^n \cdot a_1^2 + a_0^2 + 2^{n/2}\left((a_1 + a_0)^2 - a_1^2 - a_0^2\right)$

$O(n)$

Squaring in $T(n/2)$?

$a_1 + a_0 = 2 \cdot b + \varepsilon$

$(2b + \varepsilon)^2 = 4b^2 + 4b\varepsilon + \varepsilon^2$

$T(n) = 3\,T(n/2) + O(n)$.

$\Rightarrow T(n) = O(n^{\log_2 3}) \simeq O(n^{1.58})$

Solving the recurrence

$T(n) \le c \cdot n + 3T(n/2)$

$T(n) \le c \cdot n + 3 \cdot cn/2 + 3^2 \cdot T(n/4)$

$\le c \cdot n + 3 \cdot cn/2 + 3^2 \cdot cn/2^2 + 3^3 \cdot T(n/8)$

$\le cn + \frac{3}{2}cn + \frac{3^2}{2^2}cn + \cdots \frac{3^{\log n - 1}}{2^{\log n - 1}} cn + 3^{\log n} T(1)$

$$T(n) \le cn\underbrace{\frac{(3/2)^{\log n}-1}{3/2-1}} + 3^{\log n}$$

$$= 2cn \cdot n^{\log 3/2} + n^{\log 3}$$

$$= 2c\, n^{1+\log 3/2} + n^{\log 3}$$

$$= O(n^{\log_2 3}) \approx O(n^{1.585})$$

$\log 3/2 = \log 3 - \log 2$

$x^{\log y} = (2^{\log x})^{\log y} = 2^{\log x \cdot \log y} = (2^{\log y})^{\log x} = y^{\log x}$

What about multiplication?

$$a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$$

$$a \cdot b = \frac{(a+b)^2 - (a-b)^2}{4}$$

Multiplication directly (without going via squaring)

$a \times b$?

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

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

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

$$= a_1 \cdot b_1 \cdot 2^n + (a_0 b_1 + a_1 b_0) 2^{n/2} + a_0 b_0$$

HW

Can these three terms a_1b_1, $a_0b_1 + a_1b_0$, a_0b_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 $a = a_2 \cdot 2^{2n/3} + a_1 \cdot 2^{n/3} + a_0$

$$a^2 = a_2^2 \cdot 2^{4n/3} + 2a_2a_1 \cdot 2^{n} + (a_1^2 + 2a_0a_2)\, 2^{2n/3}$$
$$+ 2a_1a_0 \cdot 2^{n/3} + a_0^2 \cdot 2^0$$

$$T(n) = \alpha\, T(n/3) + O(n)$$

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

current best $n^{1.585}$

$$\log_3 6 = 1.63$$
$$\log_3 5 = 1.46 \implies O(n^{1.46})$$

Can we compute the desired terms with 5/6 squarings? + $O(n)$ operations.

easy to do with 6 squarings.

$$a_0^2,\ a_1^2,\ a_2^2,\ (a_0+a_1)^2,\ (a_0+a_2)^2,\ (a_1+a_2)^2$$

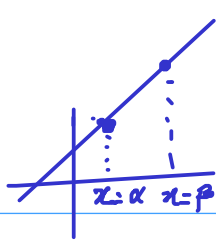
Idea:

To improve, we need to see it as a squaring of a polynomial.

$$A(x) = a_2 x^2 + a_1 x + a_0$$

$$(A(x))^2 = a_2^2 x^4 + 2a_1a_2 x^3 + (a_1^2 + 2a_0a_2) x^2$$
$$+ 2a_0a_1 x + a_0^2$$

Polynomial Representations: for degree d



→ Coefficients d+1

→ Roots d

→ Evaluations d+1

$x = \alpha_1$
$x = \alpha_2$
$x = \alpha_3$
$x = \alpha_{d+1}$

How easy/difficult it is to square a polynomial in evaluation representation?

Given evaluations of $A(x)$, computing evaluations of $A^2(x)$?

$$A^2(\alpha) = (A(\alpha))^2$$

→ Each evaluation of $A^2(x)$ needs one squaring

→ How many evaluations of $A^2(x)$ needed?

Five evaluations → five squarings.

But we are really interested in coeff of $A^2(x)$.

Plan:

coeffs of $A(x)$ $\xrightarrow{?}$ (five) Evaluations of $A(x)$

a_0, a_1, a_2

① $\xrightarrow{?}$ $O(n)$

② squaring

coeffs of $A^2(x)$ $\xleftarrow{?}$ ③ $O(n)$ Evaluations of $A^2(x)$ (five)

① $A(x) = a_2x^2 + a_1x + a_0$

$A(x=\alpha) = a_2\alpha^2 + a_1\alpha + a_0$

$a_0, a_1, a_2 \rightarrow n/3$ bits

$O(n)$

$x = 0, 1, -1, 2, -2$

$A(0) = a_0$ $\quad A(1) = a_0 + a_1 + a_2$ $\quad A(-1) = a_0 - a_1 + a_2$

$n/3+4$ bits ← $A(2) = a_0 + 2a_1 + 4a_2$

② $A(0)$, $A(1)$, $A(-1)$, $A(2)$, $A(-2)$

↓ Square

$A^2(0)$, $A^2(1)$, $A^2(-1)$, $A(2)$, $A(-2)$

$5\,T(n/3) + O(n)$

③ Define $S(x) := A^2(x)$

How are coeffs and evaluations of $S(x)$ are related?

$$S(0) = a_0^2$$

$$S(1) = a_0^2 + 2a_0a_1 + a_2^2 + 2a_0a_2 + 2a_1a_2 + a_2^2$$

$$S(2) = a_0^2 + 2 \cdot 2a_0a_1 + 4\,(a_2^2 + 2a_0a_2) + 8 \cdot 2a_1a_2 + 16\,a_2^2$$

$$\begin{bmatrix} S(0) \\ S(1) \\ S(2) \\ S(-1) \\ S(-2) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \end{bmatrix} \begin{bmatrix} a_0^2 \\ 2a_0a_1 \\ a_1^2 + 2a_0a_2 \\ 2a_1a_2 \\ a_2^2 \end{bmatrix}$$

eval-vector $= [\,M\,]_{5\times 5}$ coeff-vector

$M^{-1} \cdot$ eval-vector $=$ Coeff-vector

Can we do this in $O(n)$?

We can pre-compute M^{-1} and store it.

Once computed, M^{-1} can be used to square any integer.

M^{-1} .. eval-vector

M^{-1}: 5×5

eval-vector: $(n/3 + 4) \times 2 = O(n)$

All entries of M^{-1} are constants.

HW Multiplication/division of an n bit integer with a constant can be done in $O(n)$ bit operations.

need 25 multiplications and 20 additions.
Overall $O(n)$ time.

$$T(n) = 5\,T(n/3) + O(n)$$

$$T(n) \approx O(n^{1.46}) \quad \text{Toom-Cook}$$

Integer Multiplication. History

1960 Karatsuba $O(n^{1.585})$

Toom Cook $O(n^{1.46})$

can be further generalized by dividing the integers into more parts and get better and better time complexity.
But, the time complexity will remain something like $O(n^{1+\varepsilon})$ for $\varepsilon > 0$.

1971 Schönhage Strassen $O(n \log n \log\log n)$

2005 Fürer $O(n \log n\, 2^{\log^* n})$

2019 Harvey, Vander Hoeven $O(n \log n)$

Ideas: polynomial evaluation (also known as discrete fourier Transform)
divide and conquer and other ideas.