Algorithms M2 IF
Divide and Conquer

Michael Lampis

Fall 2019

Divide and Conquer

- Divide and Conquer is a basic algorithmic technique
  - Idea: solve a problem by breaking it down into sub-instances, solving independently, merging solutions.
  - Algorithms are usually **recursive**.
- In these slides:
  - Reminder: Binary search
  - Reminder: Mergesort
  - Master Theorem for Analyzing recurrences
  - Integer Multiplication
  - Matrix Multiplication
  - Median

Reminder of two basic algorithms: Binary Search and Mergesort

Binary Search

Problem:

- Given: **sorted** array $A[1 \ldots n]$ of n elements, specific element x.
- Question: is x in the array? If yes, at what position?
- Operations: $A[i] \stackrel{?}{<} x, A[i] \stackrel{?}{>} x, A[i] \stackrel{?}{=} x$ take $O(1)$ time.

Binary Search

Problem:

- Given: **sorted** array $A[1 \dots n]$ of n elements, specific element x.
- Question: is x in the array? If yes, at what position?
- Operations: $A[i] \stackrel{?}{<} x, A[i] \stackrel{?}{>} x, A[i] \stackrel{?}{=} x$ take $O(1)$ time.

Trivial to solve in $O(n)$ time:

- For $i = 1, \dots n$ if $A[i] = x$ then return i.
- Return NO.

Binary Search

Problem:

- Given: **sorted** array $A[1 \dots n]$ of n elements, specific element x.
- Question: is x in the array? If yes, at what position?
- Operations: $A[i] \stackrel{?}{<} x, A[i] \stackrel{?}{>} x, A[i] \stackrel{?}{=} x$ take $O(1)$ time.

Trivial to solve in $O(n)$ time:

- For $i = 1, \dots n$ if $A[i] = x$ then return i.
- Return NO.

Binary Search achieves logarithmic time:

- BinSearch(A,l,h):
  - If $l > h$ return NO
  - If $l == h$ return $(A[l] == x)?l : NO$
  - Let $m = (l + h)/2$
  - If $A[m] == x$ return m
  - If $A[m] > x$ return BinSearch($A, l, m - 1$)
  - If $A[m] < x$ return BinSearch($A, m + 1, h$)

Binary Search

- Recursive algorithm of previous slide is called with parameters $A, 1, n$.
- Idea: will search area of array from $A[l]$ to $A[h]$ (inclusive).

Binary Search

- Recursive algorithm of previous slide is called with parameters $A, 1, n$.
- Idea: will search area of array from $A[l]$ to $A[h]$ (inclusive).

Correctness:

- Proof by induction on $h - l$.
- For $h - l \leq 0$ trivial.
- For $h - l \geq 1$, $[l, m-1]$ and $[m+1, h]$ are strictly smaller intervals.
- By induction algorithm is correct for both. Because A is sorted, if x is somewhere it must be in the interval we search.

Binary Search

- Recursive algorithm of previous slide is called with parameters $A, 1, n$.
- Idea: will search area of array from $A[l]$ to $A[h]$ (inclusive).

Correctness:

- Proof by induction on $h - l$.
- For $h - l \leq 0$ trivial.
- For $h - l \geq 1$, $[l, m - 1]$ and $[m + 1, h]$ are strictly smaller intervals.
- By induction algorithm is correct for both. Because A is sorted, if x is somewhere it must be in the interval we search.

Complexity:

- Searching an interval of length $\leq 1 \rightarrow O(1)$ time.
- Searching an interval of length n takes at most $O(1)$ plus the time it takes for an interval of length $n/2$.

$$T(n) \leq T(n/2) + O(1) \leq O(\log n)$$

Mergesort

Problem:

- Given: array $A[1 \ldots n]$ of n elements.
- Question: output elements of A in increasing order (sort A).
- Operations: $A[i] \overset{?}{<} A[j], A[i] \overset{?}{>} A[j], A[i] \overset{?}{=} A[j]$, copies take $O(1)$ time.

Mergesort

Problem:

- Given: array $A[1 \dots n]$ of n elements.
- Question: output elements of A in increasing order (sort A).
- Operations: $A[i] \stackrel{?}{<} A[j], A[i] \stackrel{?}{>} A[j], A[i] \stackrel{?}{=} A[j]$, copies take $O(1)$ time.

Mergesort:

- Suppose we have a `Merge` procedure
  - `Merge` is given two **sorted** arrays A, B
  - Output: a sorted array with all the elements of A, B

`Mergesort`$(A[1 \dots n])$

- If $n \leq 10$ trivial...
- Else
  - `Mergesort`$(A[1 \dots n/2]) \rightarrow A_1$
  - `Mergesort`$(A[n/2 + 1 \dots n]) \rightarrow A_2$
  - `Merge`(A_1, A_2)

Mergesort

Correctness:

- Proof by induction on size of array n.
- If $n \leq 10$ trivial.
- By induction algorithm correctly sorts A_1, A_2. If `Merge` is correct, then the sorting is correct.

Mergesort

Correctness:

- Proof by induction on size of array n.
- If $n \leq 10$ trivial.
- By induction algorithm correctly sorts A_1, A_2. If `Merge` is correct, then the sorting is correct.

Complexity:

- Let $T(n), M(n)$ be the complexity of `Mergesort, Merge` respectively, where n is total input size.
- Then

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

- If $M(n) = O(n)$ then $T(n) = O(n \log n)$ (why?)

Merge

- Our sorting algorithm is done, except for `Merge`
- Can we merge two sorted arrays in linear time?

$\texttt{Merge}(A[1 \ldots n], B[1 \ldots m])$

- If $n = 0$ or $m = 0$ trivial
- If $A[1] < B[1]$ output $A[1]$ and then $\texttt{Merge}(A[2 \ldots n], B[1 \ldots m])$
- If $A[1] \geq B[1]$ output $B[1]$ and then $\texttt{Merge}(A[1 \ldots n], B[2 \ldots m])$

Merge

- Our sorting algorithm is done, except for `Merge`
- Can we merge two sorted arrays in linear time?

$\texttt{Merge}(A[1 \ldots n], B[1 \ldots m])$

- If $n = 0$ or $m = 0$ trivial
- If $A[1] < B[1]$ output $A[1]$ and then $\texttt{Merge}(A[2 \ldots n], B[1 \ldots m])$
- If $A[1] \geq B[1]$ output $B[1]$ and then $\texttt{Merge}(A[1 \ldots n], B[2 \ldots m])$

Correctness: easy by induction (how?)

Complexity:

$$M(n+m) \leq O(1) + M(n+m-1) = O(n+m)$$

Analysis of Divide and Conquer

Solving Recurrence Relations: Induction

- When analyzing recursive (divide&conquer) algorithms we often have to solve equations of the form:

$$T(n) \leq T(n_1) + T(n_2) + \ldots + f(n)$$

- Where:
  - $T(n)$ is the running time of the algorithm for an input of size n
  - $n_1, n_2, \ldots, < n$ (why?)
  - $f(n)$ is the running time of breaking down the problem into sub-problems and then putting the solutions back together.

Example:

$$\begin{aligned} T(n) &\leq 2T(n/2) + Cn \quad \text{(Mergesort)} \\ T(n) &\leq T(n/2) + C \quad \text{(BinSearch)} \\ T(n) &\leq \frac{1}{n}\sum_{i=0}^{n}(T(n-i) + T(i)) + Cn \quad \text{(Quicksort Avg)} \end{aligned}$$

Solving Recurrence Relations: Induction

- Solving such relations can be tricky. One approach: guess the solution, prove by induction.

Solving Recurrence Relations: Induction

- Solving such relations can be tricky. One approach: guess the solution, prove by induction.

Claim: Mergesort has $T(n) \leq Cn \log n$

$$
\begin{aligned}
T(n) &\leq 2T(n/2) + Cn \leq \\
&\leq 2C\frac{n}{2}\log\frac{n}{2} + Cn = \\
&= Cn\log n - Cn + Cn = Cn\log n
\end{aligned}
$$

Solving Recurrence Relations: Induction

- Solving such relations can be tricky. One approach: guess the solution, prove by induction.

Claim: BinSearch has $T(n) \leq C \log n$

$$\begin{aligned} T(n) &\leq T(n/2) + C \leq \\ &\leq C \log \frac{n}{2} + C = \\ &= C \log n - C + C = C \log n \end{aligned}$$

Solving Recurrence Relations: Induction

- Solving such relations can be tricky. One approach: guess the solution, prove by induction.

- Generally, this approach is tricky, because we have to “guess” and then prove the correct formula.
- It often helps to “unroll” the recurrence for a few steps to see where things are going. Example (Mergesort):

$$T(n) \leq 2T(n/2) + Cn \leq 4T(n/4) + 2C\frac{n}{2} + Cn \leq 8T(n/8) + 3Cn \ldots$$

The Master Theorem

A more standard way to handle (some) recurrence relations

- Let $T(n) = aT(n/b) + O(n^d)$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

The Master Theorem

A more standard way to handle (some) recurrence relations

- Let $T(n) = aT(n/b) + O(n^d)$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof:

$$\begin{aligned} T(n) &= a^i T(n/b^i) + n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \ldots + \left(\frac{a}{b^d}\right)^{i-1}\right) = \\ &= ? \end{aligned}$$

- If $a < b^d$ then $\rightarrow a^{\log_b n} + O(n^d) = n^{\log_b a} + O(n^d) = O(n^d)$
- If $a = b^d$ then $\rightarrow n^d\,(\log_b n) = O(n^d \log n)$
- If $a > b^d$ then $\rightarrow a^{\log_b n} + n^d \left((\frac{a}{b^d})^{\log_b n}\right) = O(n^{\log_b a})$

Integer Multiplication

Integer Multiplication

Problem:

- Input: two n-bit numbers, $A = a_{n-1}, a_{n-2}, \ldots, a_0$ and $B = b_{n-1}, b_{n-2}, \ldots, b_0$ where a_0, b_0 are the least significant bits.
- Output: the product $A \times B$

Integer Multiplication

Problem:

- Input: two n-bit numbers, $A = a_{n-1}, a_{n-2}, \ldots, a_0$ and $B = b_{n-1}, b_{n-2}, \ldots, b_0$ where a_0, b_0 are the least significant bits.
- Output: the product $A \times B$

Elementary-school algorithm relies on two observations:

- Multiplication of a number by 2^i is easy (append i times 0 at the end)
- Addition can be done in time $O(n)$
- (We of course assume that addition/multiplication of two bits is $O(1)$)

Integer Multiplication

Problem:

- Input: two n-bit numbers, $A = a_{n-1}, a_{n-2}, \ldots, a_0$ and $B = b_{n-1}, b_{n-2}, \ldots, b_0$ where a_0, b_0 are the least significant bits.
- Output: the product $A \times B$

Elementary-school algorithm relies on two observations:

- Multiplication of a number by 2^i is easy (append i times 0 at the end)
- Addition can be done in time $O(n)$
- (We of course assume that addition/multiplication of two bits is $O(1)$)

Part 1: Multiply $A = a_{n-1} \ldots a_0$ with b_i

- Can be done in $O(n)$:
- If $b_i = 0$, result is 0, otherwise result is A.

Integer Multiplication

Problem:

- Input: two n-bit numbers, $A = a_{n-1}, a_{n-2}, \ldots, a_0$ and $B = b_{n-1}, b_{n-2}, \ldots, b_0$ where a_0, b_0 are the least significant bits.
- Output: the product $A \times B$

Elementary-school algorithm relies on two observations:

- Multiplication of a number by 2^i is easy (append i times 0 at the end)
- Addition can be done in time $O(n)$
- (We of course assume that addition/multiplication of two bits is $O(1)$)

Part 1: Multiply $A = a_{n-1} \ldots a_0$ with b_i

- Can be done in $O(n)$:
- If $b_i = 0$, result is 0, otherwise result is A.

Part 2: General multiplication

- For each $i \in \{0, \ldots, n-1\}$ compute $b_i \times A \times 2^i$.
- Sum all these values.
  - n additions of $O(n)$ time each $\rightarrow O(n^2)$

Karatsuba's algorithm

- Goal: do better than $O(n^2)$ for n-bit integer multiplication
- Note: Kolmogorov conjectured in the '60s that this is impossible
    - He was wrong!

Karatsuba's algorithm

- Goal: do better than $O(n^2)$ for n-bit integer multiplication
- Note: Kolmogorov conjectured in the '60s that this is impossible
  - He was wrong!

A divide&conquer approach

- Let A_1 the number made up of the first $n/2$ bits of A, A_2 the rest. (Similarly B_1, B_2)

$$\begin{aligned} A &= A_1 2^{n/2} + A_2 \\ B &= B_1 2^{n/2} + B_2 \Rightarrow \\ A \times B &= (A_1 \times B_1) 2^n + (A_1 \times B_2 + A_2 \times B_1) 2^{n/2} + A_2 \times B_2 \end{aligned}$$

Karatsuba's algorithm

- Goal: do better than $O(n^2)$ for n-bit integer multiplication
- Note: Kolmogorov conjectured in the '60s that this is impossible
  - He was wrong!

A divide&conquer approach

- Let A_1 the number made up of the first $n/2$ bits of A, A_2 the rest. (Similarly B_1, B_2)

$$\begin{aligned} A &= A_1 2^{n/2} + A_2 \\ B &= B_1 2^{n/2} + B_2 \Rightarrow \\ A \times B &= (A_1 \times B_1) 2^n + (A_1 \times B_2 + A_2 \times B_1) 2^{n/2} + A_2 \times B_2 \end{aligned}$$

Complexity: $T(n) = 4T(n/2) + O(n)$

Karatsuba's algorithm

- Goal: do better than $O(n^2)$ for n-bit integer multiplication
- Note: Kolmogorov conjectured in the '60s that this is impossible
  - He was wrong!

A divide&conquer approach

- Let A_1 the number made up of the first $n/2$ bits of A, A_2 the rest. (Similarly B_1, B_2)

$$\begin{aligned} A &= A_1 2^{n/2} + A_2 \\ B &= B_1 2^{n/2} + B_2 \Rightarrow \\ A \times B &= (A_1 \times B_1) 2^n + (A_1 \times B_2 + A_2 \times B_1) 2^{n/2} + A_2 \times B_2 \end{aligned}$$

Complexity: $T(n) = 4T(n/2) + O(n)$
$T(n) = O(n^2)$:-(

Karatsuba's algorithm

- Karatsuba's algorithm relies on divide&conquer but:
  - Recognizes that the costly part is multiplication
  - Manages to reduce the number of multiplications by doing more additions/subtractions

Karatsuba's algorithm

- Karatsuba's algorithm relies on divide&conquer but:
  - Recognizes that the costly part is multiplication
  - Manages to reduce the number of multiplications by doing more additions/subtractions
- Recall:

$$A \times B = (A_1 \times B_1)2^n + (A_1 \times B_2 + A_2 \times B_1)2^{n/2} + A_2 \times B_2$$

Karatsuba's algorithm

- Karatsuba's algorithm relies on divide&conquer but:
  - Recognizes that the costly part is multiplication
  - Manages to reduce the number of multiplications by doing more additions/subtractions
- Recall:

$$A \times B = (A_1 \times B_1)2^n + (A_1 \times B_2 + A_2 \times B_1)2^{n/2} + A_2 \times B_2$$

Calculate:

- $A_1 \times B_1$
- $A_2 \times B_2$

Karatsuba's algorithm

- Karatsuba's algorithm relies on divide&conquer but:
  - Recognizes that the costly part is multiplication
  - Manages to reduce the number of multiplications by doing more additions/subtractions
- Recall:

$$A \times B = (A_1 \times B_1)2^n + (A_1 \times B_2 + A_2 \times B_1)2^{n/2} + A_2 \times B_2$$

Calculate:

- $A_1 \times B_1$
- $A_2 \times B_2$
- $(A_1 + A_2) \times (B_1 + B_2)$

Karatsuba's algorithm

- Karatsuba's algorithm relies on divide&conquer but:
  - Recognizes that the costly part is multiplication
  - Manages to reduce the number of multiplications by doing more additions/subtractions
- Recall:

$$A \times B = (A_1 \times B_1)2^n + (A_1 \times B_2 + A_2 \times B_1)2^{n/2} + A_2 \times B_2$$

  Calculate:
- $A_1 \times B_1$
- $A_2 \times B_2$
- $(A_1 + A_2) \times (B_1 + B_2)$
- Key idea:
  $A_1 \times B_2 + A_2 \times B_1 = (A_1 + A_2) \times (B_1 + B_2) - A_1 \times B_1 - A_2 \times B_2$

Karatsuba's algorithm – Analysis

- We perform 3 (instead of 4) multiplications of numbers with $n/2$ digits
  - Not quite true: $A_1 + A_2$ could have $n/2 + 1$ digits. Doesn't matter much for asymptotic analysis.
- We perform several additions/subtractions of n digit numbers.

Karatsuba's algorithm – Analysis

- We perform 3 (instead of 4) multiplications of numbers with $n/2$ digits
  - Not quite true: $A_1 + A_2$ could have $n/2 + 1$ digits. Doesn't matter much for asymptotic analysis.
- We perform several additions/subtractions of n digit numbers.

Complexity:
$T(n) \leq 3T(n/2) + O(n)$

Karatsuba's algorithm – Analysis

- We perform 3 (instead of 4) multiplications of numbers with $n/2$ digits
  - Not quite true: $A_1 + A_2$ could have $n/2 + 1$ digits. Doesn't matter much for asymptotic analysis.
- We perform several additions/subtractions of n digit numbers.

Complexity:
$T(n) \leq 3T(n/2) + O(n)$
Master Theorem
$a = 3, b = 2, d = 1, d < \log_b a \Rightarrow T(n) = O(n^{\log 3}) \approx O(n^{1.6}) << n^2$

Karatsuba's algorithm – Analysis

- We perform 3 (instead of 4) multiplications of numbers with $n/2$ digits
  - Not quite true: $A_1 + A_2$ could have $n/2 + 1$ digits. Doesn't matter much for asymptotic analysis.
- We perform several additions/subtractions of n digit numbers.

Complexity:
$T(n) \leq 3T(n/2) + O(n)$
Master Theorem
$a = 3, b = 2, d = 1, d < \log_b a \Rightarrow T(n) = O(n^{\log 3}) \approx O(n^{1.6}) << n^2$

Lesson: in divide&conquer, decreasing the number of sub-problems is hugely important, because their total number increases exponentially!

Matrix Multiplication

Matrix Multiplication – Easy Algorithm

Problem:

- Input: two $n \times n$ matrices A, B
- Output: the product $C = A \times B$
- Assumption: adding/multiplying two elements takes time $O(1)$

Matrix Multiplication – Easy Algorithm

Problem:

- Input: two $n \times n$ matrices A, B
- Output: the product $C = A \times B$
- Assumption: adding/multiplying two elements takes time $O(1)$

Simple algorithm:

- To calculate $C[i, j]$ we multiply row i of A with column j of B
  - For $k \in \{1, \ldots, n\}$ sum up $A[i, k] \times B[k, j]$
- $O(n)$ per element of $C \Rightarrow O(n^3)$.

Goal: achieve complexity less than $O(n^3)$.

Matrix Multiplication – Divide&Conquer

- We want to calculate $C = A \times B$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Matrix Multiplication – Divide&Conquer

- We want to calculate $C = A \times B$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Matrix Multiplication – Divide&Conquer

- We want to calculate $C = A \times B$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 multiplications of $(n/2) \times (n/2)$ matrices (additions take time $O(n^2)$)

$$T(n) = 8T(n/2) + O(n^2)$$

Matrix Multiplication – Divide&Conquer

- We want to calculate $C = A \times B$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 multiplications of $(n/2) \times (n/2)$ matrices (additions take time $O(n^2)$)

$$T(n) = 8T(n/2) + O(n^2)$$

Master Theorem $a = 8, b = 2, d = 2, \log_b a = 3 > 2 \Rightarrow O(n^{\log_b a}) = O(n^3)$

Strassen's algorithm

Need to calculate:

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

Strassen's algorithm

Need to calculate:

$$\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}$$

Will calculate:

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Strassen's algorithm

Need to calculate:

$$
\begin{aligned}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{aligned}
$$

Will calculate:

????

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Strassen's algorithm

- Easy but tedious to verify that:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Strassen's algorithm

- Easy but tedious to verify that:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

- We calculate C with 7 multiplications of $(n/2) \times (n/2)$ matrices (and some additions)
- Complexity: $O(n^{\log 7}) \approx O(n^{2.81}) << n^3$

Median

Median

Reminder:

- Input: unsorted array $A[1 \dots n]$.
- Output: median element (element which would be in $A[n/2]$ in sorted array)

- Easy $O(n \log n)$ (sort)
- We have seen $O(n)$ **randomized**
- Goal: $O(n)$ deterministic

Will solve a more general problem: given A, k, return the k-th smallest element.

Median – Given good pivot

First idea: suppose that it's easy to find a number p "close" to the median.

- Partition A into L, R, elements smaller, larger than p respectively.
- If $|L| \geq k$ then we solve the same problem in L
- If $|L| < k$ then we solve the problem in $A \setminus L$ for $k' = k - |L|$

Median – Given good pivot

First idea: suppose that it's easy to find a number p "close" to the median.

- Partition A into L, R, elements smaller, larger than p respectively.
- If $|L| \geq k$ then we solve the same problem in L
- If $|L| < k$ then we solve the problem in $A \setminus L$ for $k' = k - |L|$

Suppose that $\min(|L|, |R|) \geq n/3$.

$$T(n) \leq T(2n/3) + O(n)$$

Median – Given good pivot

First idea: suppose that it's easy to find a number p "close" to the median.

- Partition A into L, R, elements smaller, larger than p respectively.
- If $|L| \geq k$ then we solve the same problem in L
- If $|L| < k$ then we solve the problem in $A \setminus L$ for $k' = k - |L|$

Suppose that $\min(|L|, |R|) \geq n/3$.

$$T(n) \leq T(2n/3) + O(n)$$

- This gives $T(n) = O(n)$.
- If we find a good pivot problem is easy!
- How do we find it?
  - Best pivot is the median.
  - This is the same as the original problem.
  - We must find it in sub-linear time! (otherwise we'll get $O(n \log n)$)

Find a good pivot

Idea: median is a good pivot!

- We will find the median of a much smaller array.
- Partition A into groups of 5 elements.
- Sort each group
- Let B be the array that contains the median of each group
  - $|B| = n/5$
- Find the median of B (recurse!). Let p be that number.
- Use algorithm of previous slide with p as pivot.

Analysis

- Key observation: p is always a pretty good pivot
  - p is bigger than $3n/10$ elements and smaller than $3n/10$ elements of A (why?)
  - $\Rightarrow \max(|L|, |R|) \leq 7n/10$

Analysis

- Key observation: p is always a pretty good pivot
  - p is bigger than $3n/10$ elements and smaller than $3n/10$ elements of A (why?)
  - $\Rightarrow \max(|L|, |R|) \leq 7n/10$

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

Analysis

- Key observation: p is always a pretty good pivot
  - p is bigger than $3n/10$ elements and smaller than $3n/10$ elements of A (why?)
  - $\Rightarrow \max(|L|, |R|) \leq 7n/10$

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

Master Theorem doesn't work!

Analysis

- Key observation: p is always a pretty good pivot
  - p is bigger than $3n/10$ elements and smaller than $3n/10$ elements of A (why?)
  - $\Rightarrow \max(|L|, |R|) \leq 7n/10$

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

Master Theorem doesn't work!
But by induction we can prove that $T(n) = O(n)$.

- Intuition $1/5 + 7/10 < 1$, so the total size of subproblems increases exponentially fast, hence $O(n)$ term dominates.