CSE 431/531: Algorithm Analysis and Design (Fall 2024)
Divide-and-Conquer

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance

Summary: Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
- Write down recurrence for running time
- Solve recurrence using master theorem

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Selection problem:
  $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Selection problem:
  $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$
- Polynomial Multiplication:
  $T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3})$

Summary: Divide-and-Conquer

- Merge sort, quicksort, count-inversions:
  $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \lg n)$
- Selection problem:
  $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$
- Polynomial Multiplication:
  $T(n) = 3T(n/2) + O(n) \Rightarrow T(n) = O(n^{\lg_2 3})$
- To improve running time, design better algorithm for "combine" step, or reduce number of recursions, ...

- Modular Exponentiation Problem
- Matrix multiplication
- Closest pair
- Convex hull

Modular Exponentiation Problem

Input: integer a, n and m

Output: $a^n \mod m$

Formula: $(A \times B) \mod m = [(A \mod m) \times (B \mod m)] \mod m$

ModExp(a, n, m)

1: **if** $n = 1$ **then return** $a \mod m$
2: $M_L \leftarrow$ ModExp$(a, \lfloor n/2 \rfloor, m)$
3: **if** n is odd **then return** $M_R \leftarrow (M_L \times (a \mod m)) \mod m$
4: **return** $(M_L \times M_R) \mod m$

Modular Exponentiation Problem

Input: integer a, n and m

Output: $a^n \mod m$

Formula: $(A \times B) \mod m = [(A \mod m) \times (B \mod m)] \mod m$

ModExp(a, n, m)

if $n = 1$ **then return** $a \mod m$
$M_L \leftarrow$ ModExp$(a, \lfloor n/2 \rfloor, m)$
if n is odd **then return** $M_R \leftarrow (M_L \times (a \mod m)) \mod m$
return $(M_L \times M_R) \mod m$

- Recurrence for ModExp: $T(n) = T(n/2) + O(1)$

Modular Exponentiation Problem

Input: integer a, n and m

Output: $a^n \mod m$

Formula: $(A \times B) \mod m = [(A \mod m) \times (B \mod m)] \mod m$

ModExp(a, n, m)

```
if n = 1 then return a mod m
M_L ← ModExp(a, ⌊n/2⌋, m)
if n is odd then return M_R ← (M_L × (a mod m)) mod m
return (M_L × M_R) mod m
```

- Recurrence for ModExp: $T(n) = T(n/2) + O(1)$
- Solving recurrence: $T(n) = O(\log n)$

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Naive Algorithm: matrix-multiplication(A, B, n)

for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n **do**
        $C[i, j] \leftarrow 0$
        for $k \leftarrow 1$ to n **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
return C

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

Input: two $n \times n$ matrices A and B

Output: $C = AB$

Naive Algorithm: matrix-multiplication(A, B, n)

for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to n **do**
    $C[i, j] \leftarrow 0$
    for $k \leftarrow 1$ to n **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
return C

- running time $= O(n^3)$

Try to Use Divide-and-Conquer

$$A = \begin{array}{|c|c|}\hline A_{11} & \overbrace{A_{12}}^{n/2} \\ \hline A_{21} & A_{22} \\ \hline \end{array}\left.\vphantom{A_{12}}\right\}n/2 \qquad B = \begin{array}{|c|c|}\hline B_{11} & \overbrace{B_{12}}^{n/2} \\ \hline B_{21} & B_{22} \\ \hline \end{array}\left.\vphantom{B_{12}}\right\}n/2$$

- $C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$
- matrix_multiplication(A, B) recursively calls matrix_multiplication(A_{11}, B_{11}), matrix_multiplication(A_{12}, B_{21}), $\cdots$

Try to Use Divide-and-Conquer

$$A = \begin{array}{|c|c|l} \multicolumn{1}{c}{} & \multicolumn{1}{c}{\overbrace{\hphantom{A_{12}}}^{n/2}} & \\ \hline A_{11} & A_{12} & \left.\vphantom{A_{12}}\right\} n/2 \\ \hline A_{21} & A_{22} & \\ \hline \end{array} \qquad B = \begin{array}{|c|c|l} \multicolumn{1}{c}{} & \multicolumn{1}{c}{\overbrace{\hphantom{B_{12}}}^{n/2}} & \\ \hline B_{11} & B_{12} & \left.\vphantom{B_{12}}\right\} n/2 \\ \hline B_{21} & B_{22} & \\ \hline \end{array}$$

- $C = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$
- matrix_multiplication(A, B) recursively calls matrix_multiplication(A_{11}, B_{11}), matrix_multiplication(A_{12}, B_{21}), $\cdots$
- Recurrence for running time: $T(n) = 8T(n/2) + O(n^2)$
- $T(n) = O(n^3)$

Strassen's Algorithm

- $T(n) = 8T(n/2) + O(n^2)$
- Strassen's Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: $T(n) = 7T(n/2) + O(n^2)$

Strassen's Algorithm

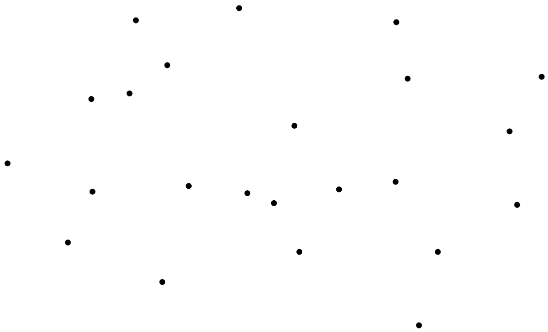
- $T(n) = 8T(n/2) + O(n^2)$
- Strassen's Algorithm: improve the number of multiplications from 8 to 7!
- New recurrence: $T(n) = 7T(n/2) + O(n^2)$
- Solving Recurrence $T(n) = O(n^{\log_2 7}) = O(n^{2.808})$

See: `https://en.wikipedia.org/wiki/Strassen_algorithm`

Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

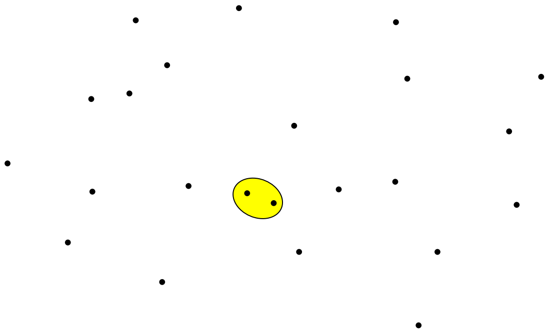
Output: the pair of points that are closest



Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

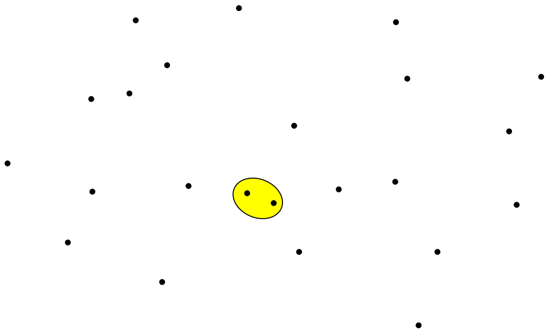
Output: the pair of points that are closest



Closest Pair

Input: n points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

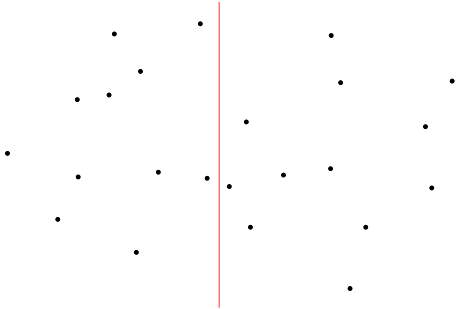
Output: the pair of points that are closest



- Trivial algorithm: $O(n^2)$ running time

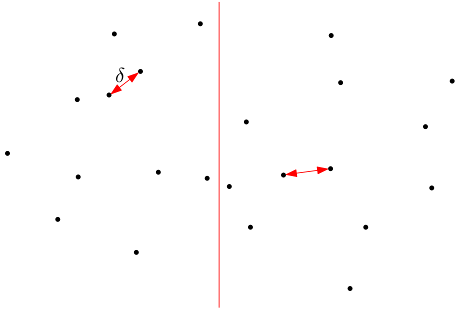
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line



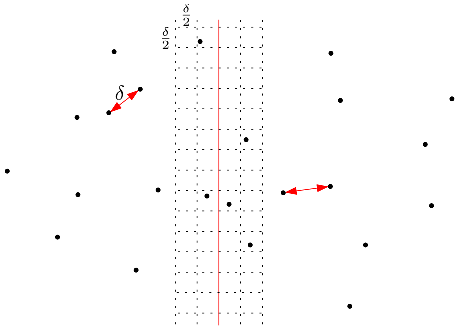
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively

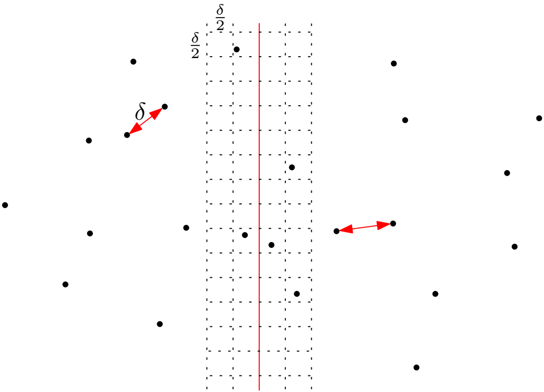


Divide-and-Conquer Algorithm for Closest Pair

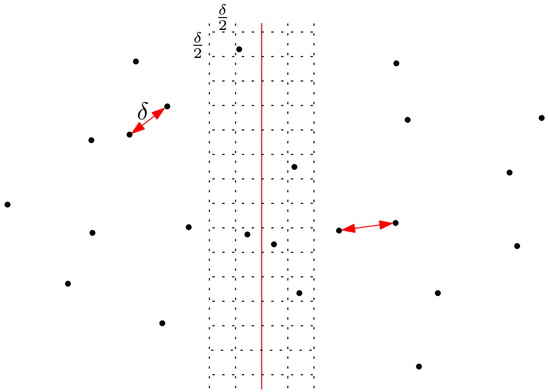
- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half



Divide-and-Conquer Algorithm for Closest Pair

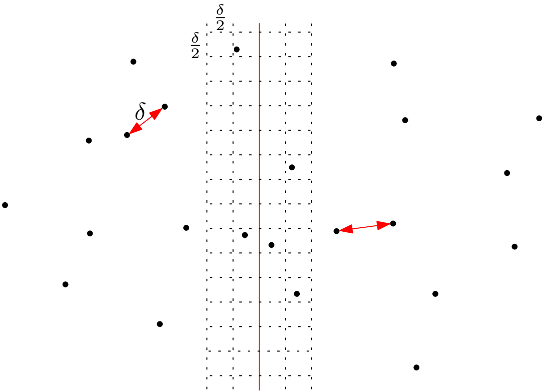


Divide-and-Conquer Algorithm for Closest Pair



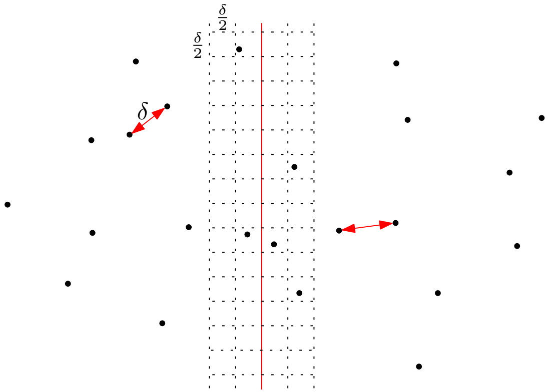
- Each box contains at most one pair

Divide-and-Conquer Algorithm for Closest Pair



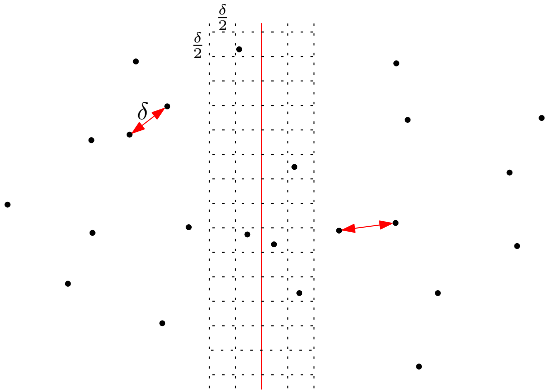
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby

Divide-and-Conquer Algorithm for Closest Pair



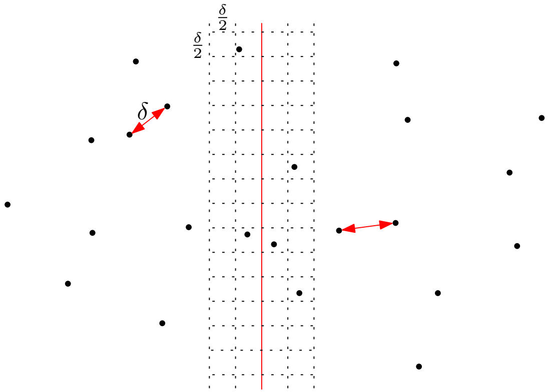
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)

Divide-and-Conquer Algorithm for Closest Pair



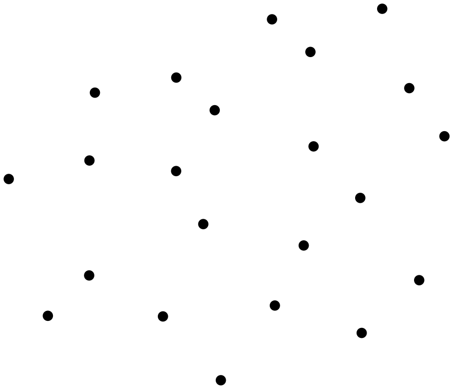
- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$

Divide-and-Conquer Algorithm for Closest Pair

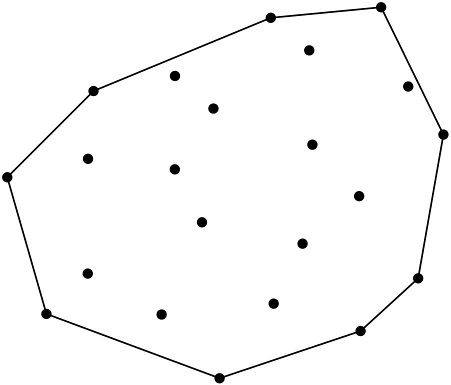


- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- time for combine $= O(n)$ (many technicalities omitted)
- Recurrence: $T(n) = 2T(n/2) + O(n)$
- Running time: $O(n \lg n)$

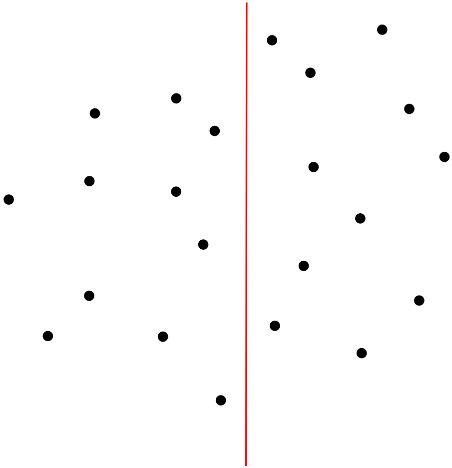
$O(n \lg n)$-Time Algorithm for Convex Hull



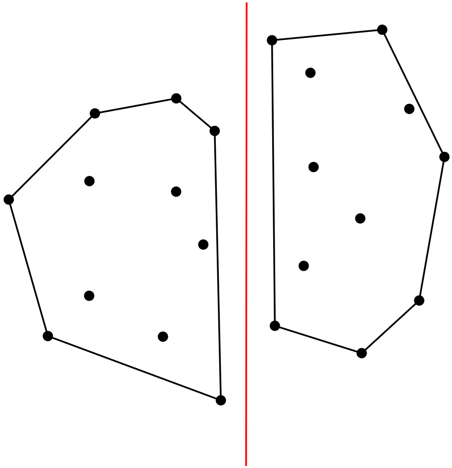
$O(n \lg n)$-Time Algorithm for Convex Hull



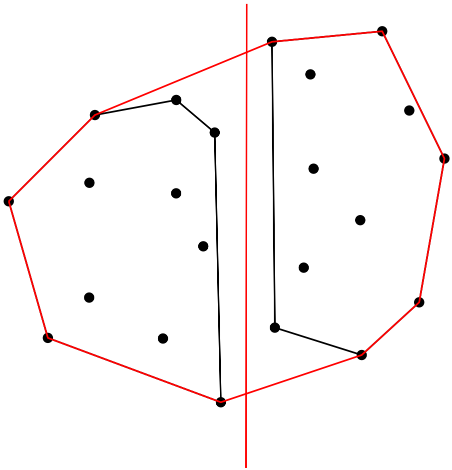
$O(n \lg n)$-Time Algorithm for Convex Hull



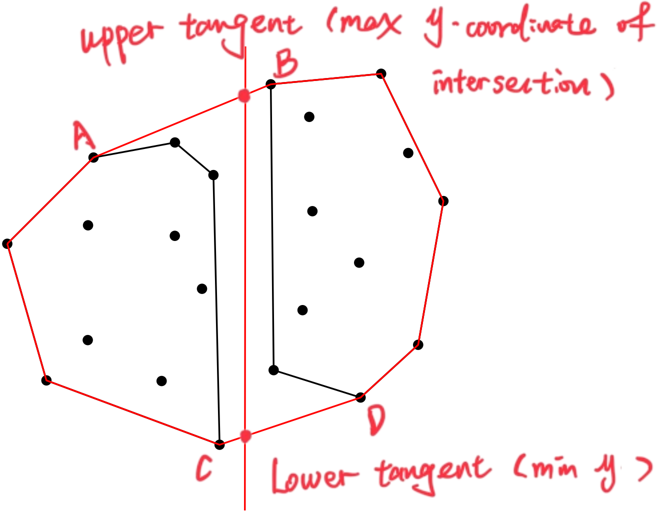
$O(n \lg n)$-Time Algorithm for Convex Hull



$O(n \lg n)$-Time Algorithm for Convex Hull



$O(n \lg n)$-Time Algorithm for Convex Hull



Outline

Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots$

n-th Fibonacci Number

Input: integer $n > 0$

Output: F_n

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

```
if n = 0 return 0
if n = 1 return 1
return Fib(n − 1) + Fib(n − 2)
```

Q: Is the running time of the algorithm polynomial or exponential in n?

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1: if $n = 0$ return 0
2: if $n = 1$ return 1
3: return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1: if $n = 0$ return 0
2: if $n = 1$ return 1
3: return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

- Running time is at least $\Omega(F_n)$

Computing F_n : Stupid Divide-and-Conquer Algorithm

Fib(n)

1: if $n = 0$ return 0
2: if $n = 1$ return 1
3: return Fib$(n-1)$ + Fib$(n-2)$

Q: Is the running time of the algorithm polynomial or exponential in n?

A: Exponential

- Running time is at least $\Omega(F_n)$
- F_n is exponential in n

Computing F_n: Reasonable Algorithm

Fib(n)

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- Dynamic Programming

Computing F_n: Reasonable Algorithm

Fib(n)

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- Dynamic Programming
- Running time = ?

Computing F_n: Reasonable Algorithm

Fib(n)

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- Dynamic Programming
- Running time $= O(n)$

Computing F_n: Even Better Algorithm

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2 \begin{pmatrix} F_{n-2} \\ F_{n-3} \end{pmatrix}$$

$$\cdots$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

power(n)

if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$R \leftarrow$ power($\lfloor n/2 \rfloor$)
$R \leftarrow R \times R$
if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
return R

Fib(n)

if $n = 0$ then return 0
$M \leftarrow$ power($n - 1$)
return $M[1][1]$

power(n)

if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$R \leftarrow$ power($\lfloor n/2 \rfloor$)
$R \leftarrow R \times R$
if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
return R

Fib(n)

if $n = 0$ then return 0
$M \leftarrow$ power($n - 1$)
return $M[1][1]$

- Recurrence for running time?

power(n)

if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$R \leftarrow$ power($\lfloor n/2 \rfloor$)
$R \leftarrow R \times R$
if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
return R

Fib(n)

if $n = 0$ then return 0
$M \leftarrow$ power($n - 1$)
return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$

power(n)

if $n = 0$ then return $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
$R \leftarrow$ power($\lfloor n/2 \rfloor$)
$R \leftarrow R \times R$
if n is odd then $R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
return R

Fib(n)

if $n = 0$ then return 0
$M \leftarrow$ power($n-1$)
return $M[1][1]$

- Recurrence for running time? $T(n) = T(n/2) + O(1)$
- $T(n) = O(\lg n)$

Running time $= O(\lg n)$: We Cheated!

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time
- Even printing $F(n)$ requires time much larger than $O(\lg n)$

Running time $= O(\lg n)$: We Cheated!

Q: How many bits do we need to represent $F(n)$?

A: $\Theta(n)$

- We can not add (or multiply) two integers of $\Theta(n)$ bits in $O(1)$ time
- Even printing $F(n)$ requires time much larger than $O(\lg n)$

Fixing the Problem

To compute F_n, we need $O(\lg n)$ basic arithmetic operations on integers