CS 577 - Divide and Conquer. Applications

Manolis Vlatakis

Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Divide and Conquer

Divide and Conquer (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Divide and Conquer (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Tendencies of DC

- Naturally recursive solutions
- Solving complexities often involve recurrences.
- Often used to improve efficiency of efficient solutions, e.g. $O(n^2) \to O(n \log n)$.
- Used in conjunction with other techniques.

Fast Exponentiation

Exponentiation by Squaring

Problem

Compute x^n where x is an integer number and n is a non-negative integer, minimizing the number of multiplications.

Let's assume that multiplication counts per one step*

**Note:* In the real world, as numbers grow larger, the cost of multiplication increases significantly. For example, $2 \times 3 = 6$, but $1,234 \times 56,789 \approx 70$ *million.*

Exponentiation by Squaring

Problem

Compute x^n where x is an integer number and n is a non-negative integer, minimizing the number of multiplications.

Let's assume that multiplication counts per one step*

🤔What is the complexity of the naive method $x^n = \underbrace{x \cdot x \cdots x}_{n \text{times}}$?

Exponentiation by Squaring

Problem

Compute x^n where x is an integer number and n is a non-negative integer, minimizing the number of multiplications.

Let's assume that multiplication counts per one step*

🤔What is the complexity of the naive method $x^n = \underbrace{x \cdot x \cdots x}_{n \text{times}}$?

$O(n)$.

Divide & Conquer Approach v1

🤔Discussion: Suggest how to divide the problem.

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

🤔How many recursive calls?

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

🤔How many recursive calls? 1 for each case.

Divide & Conquer Approach v1

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

🤔How many recursive calls? 1 for each case. Cost per call? $O(1)$ for each multiplication.

Divide & Conquer Approach v2

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Example: Fast Exponentiation Algorithm

Divide & Conquer Approach v2

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Example: Fast Exponentiation Algorithm

- Base Case: If $n = 0$, return 1.

Divide & Conquer Approach v2

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Example: Fast Exponentiation Algorithm

- Base Case: If $n = 0$, return 1.
- Recursive Case:
  - If n is even, return $\texttt{fastExp}(x, n/2)^2$.
  - If n is odd, return $x \cdot \texttt{fastExp}(x, n-1)$.

Divide & Conquer Approach v2

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Example: Fast Exponentiation Algorithm

- Base Case: If $n = 0$, return 1.
- Recursive Case:
  - If n is even, return $\texttt{fastExp}(x, n/2)^2$.
  - If n is odd, return $x \cdot \texttt{fastExp}(x, n-1)$.
- Combine: The result is computed as the recursion unwinds.

Divide & Conquer Approach v2

Reducing the problem

Consider the following cases for n:

- If n is even, $x^n = (x^{n/2})^2$.
- If n is odd, $x^n = x \cdot x^{n-1}$.

Example: Fast Exponentiation Algorithm

- Base Case: If $n = 0$, return 1.
- Recursive Case:
  - If n is even, return $\texttt{fastExp}(x, n/2)^2$.
  - If n is odd, return $x \cdot \texttt{fastExp}(x, n-1)$.
- Combine: The result is computed as the recursion unwinds.
- Recurrence: $T(n) \leq T(n/2) + O(1) = O(\log n)$.

Integer Multiplication

Integer Multiplication

Partial Product Method:

$$
\begin{array}{r}
12 \\
\times\, 13 \\
\hline
36 \\
12 \\
\hline
156
\end{array}
\qquad\qquad
\begin{array}{r}
1100 \\
\times\, 1101 \\
\hline
1100 \\
0000 \\
1100 \\
1100 \\
\hline
10011100
\end{array}
$$

Problem

Multiple two n-length binary numbers x and y, counting every bitwise operation.

Integer Multiplication

Partial Product Method:

$$
\begin{array}{r}
12 \\
\times\, 13 \\
\hline
36 \\
12 \\
\hline
156
\end{array}
\qquad
\begin{array}{r}
1100 \\
\times\, 1101 \\
\hline
1100 \\
0000 \\
1100 \\
1100 \\
\hline
10011100
\end{array}
$$

Problem

Multiple two n-length binary numbers x and y, counting every bitwise operation.

🤔What is the complexity of the partial product method?

Integer Multiplication

Partial Product Method:

$$
\begin{array}{r}
12 \\
\times\, 13 \\
\hline
36 \\
12 \\
\hline
156
\end{array}
\qquad\qquad
\begin{array}{r}
1100 \\
\times\, 1101 \\
\hline
1100 \\
0000 \\
1100 \\
1100 \\
\hline
10011100
\end{array}
$$

Problem

Multiple two n-length binary numbers x and y, counting every bitwise operation.

What is the complexity of the partial product method? $O(n^2)$.

Divide & Conquer v1

🤔Discussion : Suggest how to divide the problem.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

In decimal system:

$$x = 12 \cdot 10^2 + 34 = 1200 + 34 = 1234$$

$$y = 56 \cdot 10^2 + 78 = 5600 + 78 = 5678$$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$
\begin{aligned}
xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\
&= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0
\end{aligned}
$$

🤔How many recursive calls? 4.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call?

$$\begin{cases} x_1, y_1 \text{ are the } n/2 \text{ highest digits of } x, y \\ x_0, y_0 \text{ are the } n/2 \text{ lowest digits of } x, y \\ x_1y_1, x_1y_0, x_0y_1, x_0y_0 \text{ are } n\text{-digits numbers} \\ \alpha \cdot 2^k \text{ can be done by shifting } k \text{ digits} \\ \text{Summation of } \ell\text{-digit numbers requires } O(\ell) \text{ bit-operations.} \end{cases}$$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call? $O(n)$

🤔What is the size of the subproblem in recursive calls?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call? $O(n)$

🤔What is the size of the subproblem in recursive calls? $n/2$.

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call? $O(n)$

🤔What is the size of the subproblem in recursive calls? $n/2$.

🤔What is the recurrence?

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call? $O(n)$

🤔What is the size of the subproblem in recursive calls? $n/2$.

🤔What is the recurrence?

$$T(n) \le 4T(n/2) + cn$$

Divide & Conquer v1

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

🤔How many recursive calls? 4. Cost per call? $O(n)$

🤔What is the size of the subproblem in recursive calls? $n/2$.

🤔What is the recurrence?

$$T(n) \leq 4T(n/2) + cn = O\left(n^{\lg 4}\right) = O\left(n^2\right)$$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0y_0 := \texttt{intMult}(x_0, y_0)$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1 y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0 y_0 := \texttt{intMult}(x_0, y_0)$
- Combine: Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$

Divide & Conquer v2

High and low bits

Consider $x = x_1 \cdot 2^{n/2} + x_0$ and $y = y_1 \cdot 2^{n/2} + y_0$.

$$\begin{aligned} xy &= (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0 \end{aligned}$$

Hint: $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.

Exercise: Design an algorithm with 3 Recursive Calls

- Recursions:
  - $p := \texttt{intMult}(x_1 + x_0, y_1 + y_0)$
  - $x_1 y_1 := \texttt{intMult}(x_1, y_1)$
  - $x_0 y_0 := \texttt{intMult}(x_0, y_0)$
- Combine: Return $x_1 y_1 \cdot 2^n + (p - x_1 y_1 - x_0 y_0) \cdot 2^{n/2} + x_0 y_0$
- Recurrence: $T(n) \leq 3T(n/2) + O(n) = O\left(n^{\lg 3}\right) = O\left(n^{1.59}\right)$

Matrix Multiplication

Matrix Multiplication

Problem

Multiple two *nxn* matrices, A and B. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 2 & 1 \cdot 3 + 2 \cdot 1 \\ 3 \cdot 4 + 4 \cdot 2 & 3 \cdot 3 + 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Matrix Multiplication

Problem

Multiple two nxn matrices, A and B. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 4 + 2 \cdot 2 & 1 \cdot 3 + 2 \cdot 1 \\ 3 \cdot 4 + 4 \cdot 2 & 3 \cdot 3 + 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Algorithm: Naïve Method

```
for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            C[i][j]+ = A[i][k] · B[k][j]
        end
    end
end
```

What is the complexity of the Naïve Method?

Matrix Multiplication

Problem

Multiple two *nxn* matrices, *A* and *B*. Let $C = AB$.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 1\cdot 4 + 2\cdot 2 & 1\cdot 3 + 2\cdot 1 \\ 3\cdot 4 + 4\cdot 2 & 3\cdot 3 + 4\cdot 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

Algorithm: Naïve Method

```
for i ← 1 to n do
    for j ← 1 to n do
        for k ← 1 to n do
            C[i][j] += A[i][k] · B[k][j]
        end
    end
end
```

🤔What is the complexity of the Naïve Method? $O(n^3)$.

Divide & Conquer v1

🤔Discussion !!!: Suggest how to divide the problem.

🤔Our standard D&C questions:

Divide & Conquer v1

🤔Discussion !!!: Suggest how to divide the problem.

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

🤔Our standard D&C questions:

- 🤔 How many recursive calls?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- 🤔 What is the size of the recursive calls?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- 🤔 What is the recurrence?

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae+bg & af+bh \\ \hline ce+dg & cf+dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 8T(n/2) + cn^2$$

Divide & Conquer v1

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} ae + bg & af + bh \\ \hline ce + dg & cf + dh \end{array}\right]$$

🤔Our standard D&C questions:

- How many recursive calls? 8.
- Cost per call? $O(n^2)$ time per addition
- What is the size of the recursive calls? $n/2$.
- What is the recurrence?

$$T(n) \leq 8T(n/2) + cn^2 = O\left(n^{\lg 8}\right) = O\left(n^3\right)$$

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

🤔 What is the recurrence?

Divide & Conquer v2

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Strassen's Method (1969)

- $p_1 := a(f - h)$
- $p_2 := (a + b)h$
- $p_3 := (c + d)e$
- $p_4 := d(g - e)$
- $p_5 := (a + d)(e + h)$
- $p_6 := (b - d)(g + h)$
- $p_7 := (a - c)(e + f)$

What is the recurrence?

$$T(n) \leq 7T(n/2) + cn^2 = O\left(n^{\lg 7}\right) = O\left(n^{2.8074}\right)$$

Divide & Conquer v2

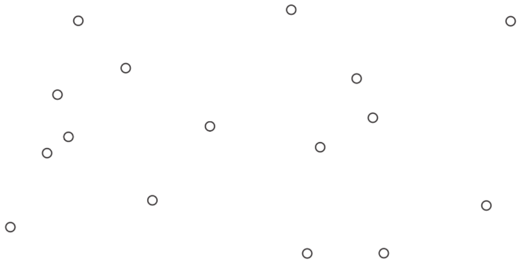
$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right]\left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ \hline p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{array}\right]$$

Current Champ: $O(n^{2.373})$

Virginia Vassilevska Williams, MIT

Closest Pairs

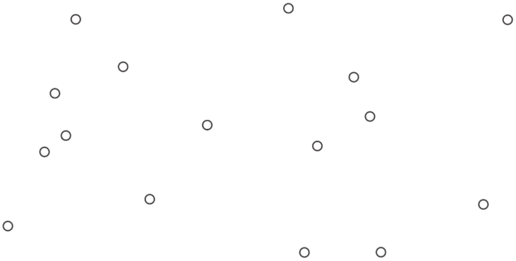
Finding the Closes Pair of Points



Problem

Given a set of n points, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, in the plane. Find the closest pair. That is, solve $\arg\min_{(p_i, p_j) \in \mathcal{P}} \{d(p_i, p_j)\}$, where $d(\cdot, \cdot)$ is the Euclidean distance.

Finding the Closes Pair of Points



Problem

Given a set of n points, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, in the plane. Find the closest pair. That is, solve $\arg\min_{(p_i, p_j) \in \mathcal{P}} \{d(p_i, p_j)\}$, where $d(\cdot, \cdot)$ is the Euclidean distance.

What is the $O(n^2)$ solution?

1-d Version

1-d Closest Pair

1-d Version

1-d Closest Pair

The points are on the line.

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points ($O(n \log n)$).

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points ($O(n \log n)$).
- Walk through sorted points and find minimum pair

1-d Version

1-d Closest Pair

The points are on the line.

$O(n \log n)$ for 1-d Closest Pair

- Sort the points ($O(n \log n)$).
- Walk through sorted points and find minimum pair ($O(n)$).

2-d Closest Pair

Divide and Conquer

1. Divide: Split point set (in half?).

2-d Closest Pair

Divide and Conquer

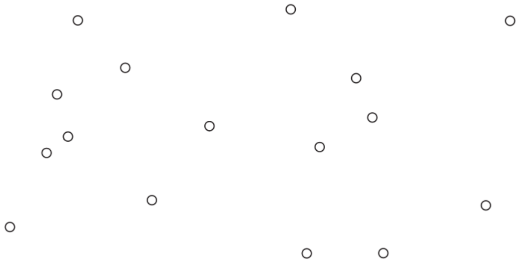
1. Divide: Split point set (in half?).
2. Conquer: Find closest pair in each partition.

2-d Closest Pair

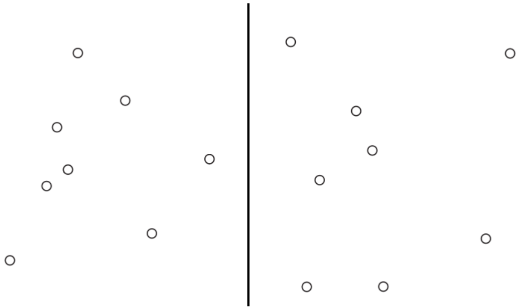
Divide and Conquer

1. Divide: Split point set (in half?).
2. Conquer: Find closest pair in each partition.
3. Combine: Merge the solutions.

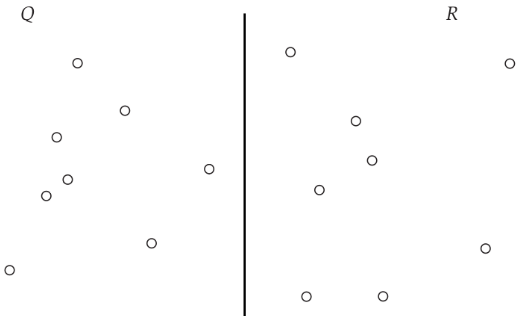
1. Divide: Split the Points



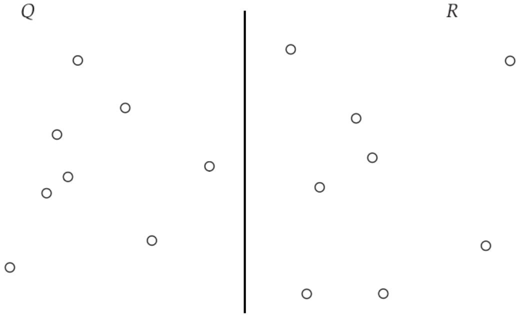
1. Divide: Split the Points



1. Divide: Split the Points



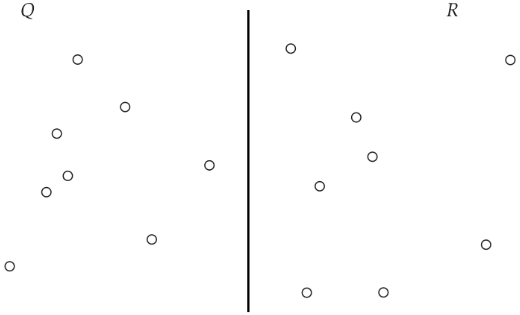
1. Divide: Split the Points



Definitions

- $\mathcal{P}_x$: Points sorted by x-coordinate.
- $\mathcal{P}_y$: Points sorted by y-coordinate.
- Q (resp. R) is left (resp. right) half of $\mathcal{P}_x$.

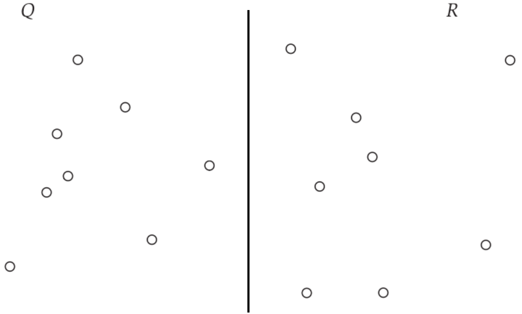
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_y without resorting.

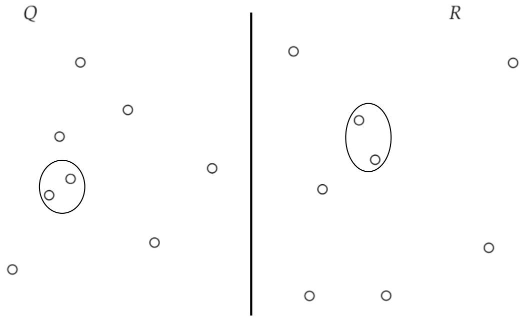
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_y without resorting.
- Running time for this:

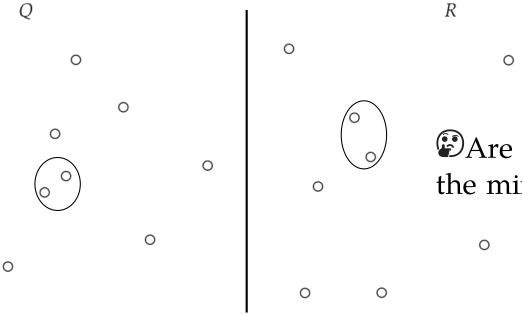
2. Conquer: Find the min in Q and R



Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_y without resorting.
- Running time for this: $O(n)$.
- Let (q_0^*, q_1^*) and (r_0^*, r_1^*) be closest pairs in Q and R.

3. Combine the Solutions.

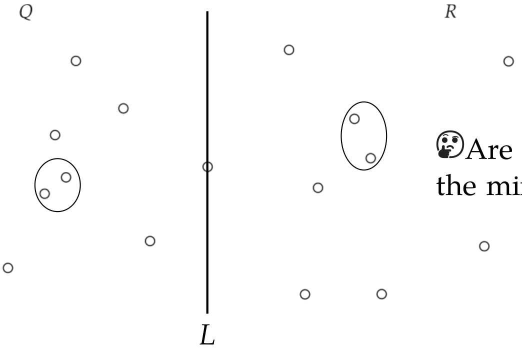


🤔Are one of these always the minimum of $\mathcal{P}$?

Key Observations

- From $\mathcal{P}_x$ and $\mathcal{P}_y$: We can create Q_x, Q_y, R_x, R_y without resorting.
- Running time for this:
- Let (q_0^*, q_1^*) and (r_0^*, r_1^*) be closest pairs in Q and R.

3. Combine the Solutions.

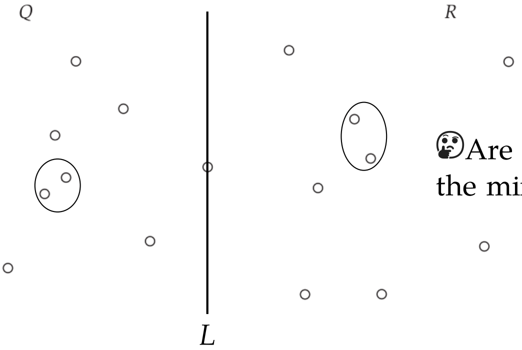


🤔Are one of these always the minimum of $\mathcal{P}$?

Claim 1

🤔 *Let* $\delta := \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$*. If there exists a* $q \in Q$ *and an* $r \in R$ *for which* $d(q, r) < \delta$*, then each of q and r are within* □ *of L.*

3. Combine the Solutions.

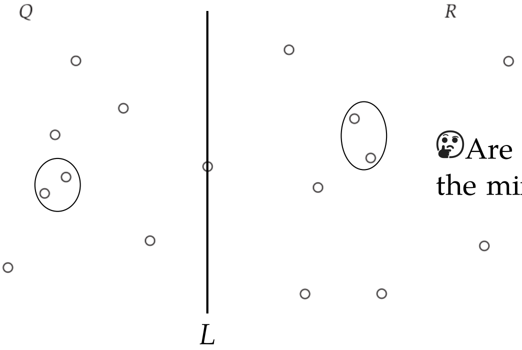


🤔Are one of these always the minimum of $\mathcal{P}$?

Claim 1

Let $\delta := \min\{d(q_0^*, q_1^*), d(r_0^*, r_1^*)\}$. If there exists a $q \in Q$ and an $r \in R$ for which $d(q, r) < \delta$, then each of q and r are within δ of L.

3. Combine the Solutions.



🤔Are one of these always the minimum of $\mathcal{P}$?

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

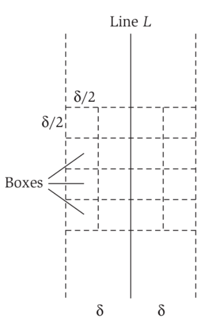
Proof.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.

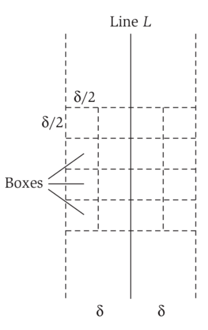


3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



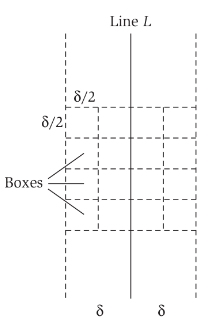
- Partition δ-space around L into $\delta/2$ squares.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



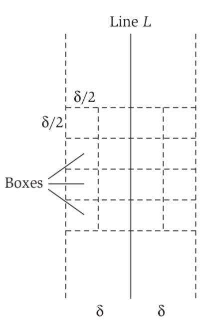
- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



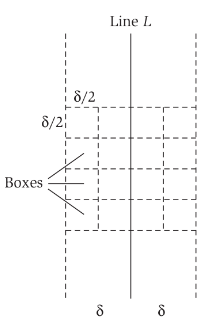
- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.
- By way of contradiction, say $d(s, s') < \delta$ and s and s' separated by 16 positions.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Proof.



- Partition δ-space around L into $\delta/2$ squares.
- At most 1 point per square else contradicts definition of δ.
- By way of contradiction, say $d(s, s') < \delta$ and s and s' separated by 16 positions.
- By counting argument, s and s' are separated by 3 rows which is at least $3\delta/2$. □

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.

3. Combine the Solutions.

Lemma 1

Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.
  - For each $p \in S$, check the distance to each of next 15 points in S_y.

3. Combine the Solutions.

Lemma 1

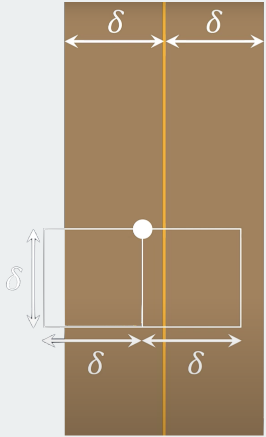
Let S be the set of points within δ of L. If there exists a $s, s' \in S$ and $d(s, s') < \delta$, then s and s' are within 15 positions of each other in S_y.

Completing the Algorithm

- Find the min pair (s, s') in S.
    - For each $p \in S$, check the distance to each of next 15 points in S_y.
- If $d(s, s') < \delta$, return (s, s')
- else return min of (q_0^*, q_1^*) and (r_0^*, r_1^*).

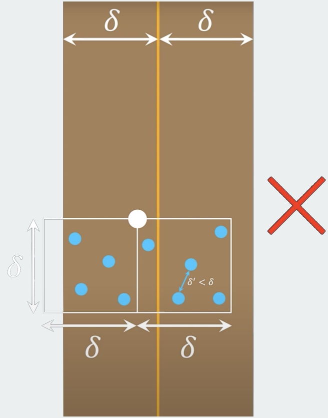
Is 15 elements optimal?

Geometry Fun Question



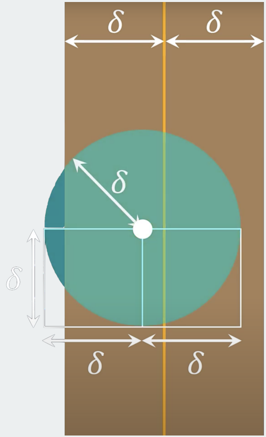
Is 15 elements optimal?

Geometry Fun Question



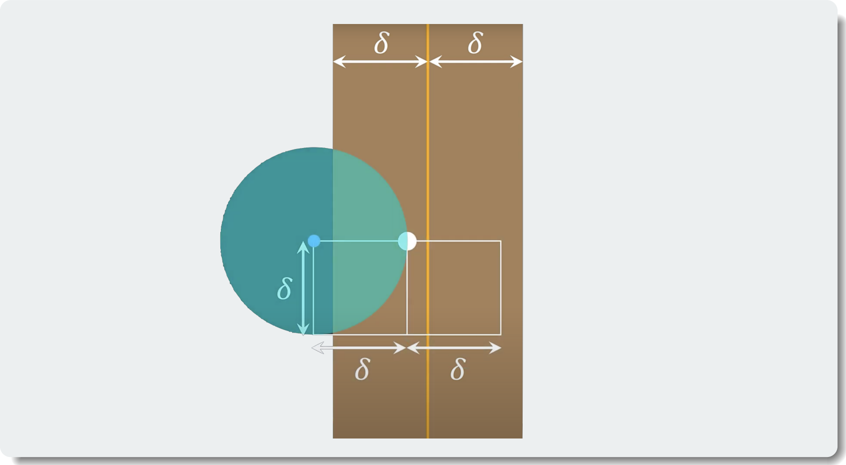
Is 15 elements optimal?

Geometry Fun Question



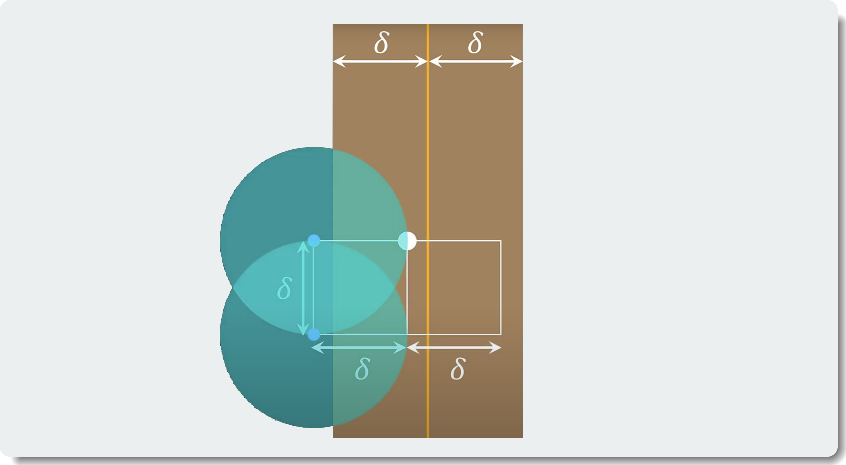
Is 15 elements optimal?

Geometry Fun Question



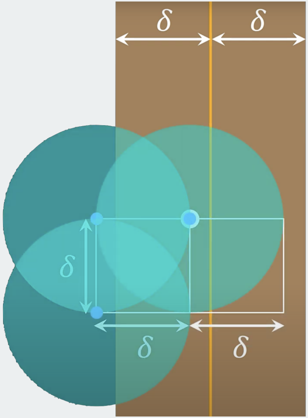
Is 15 elements optimal?

Geometry Fun Question



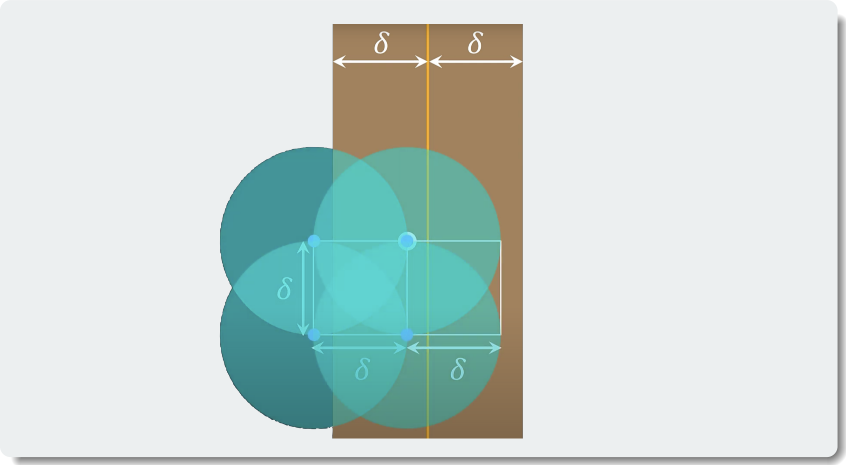
Is 15 elements optimal?

Geometry Fun Question



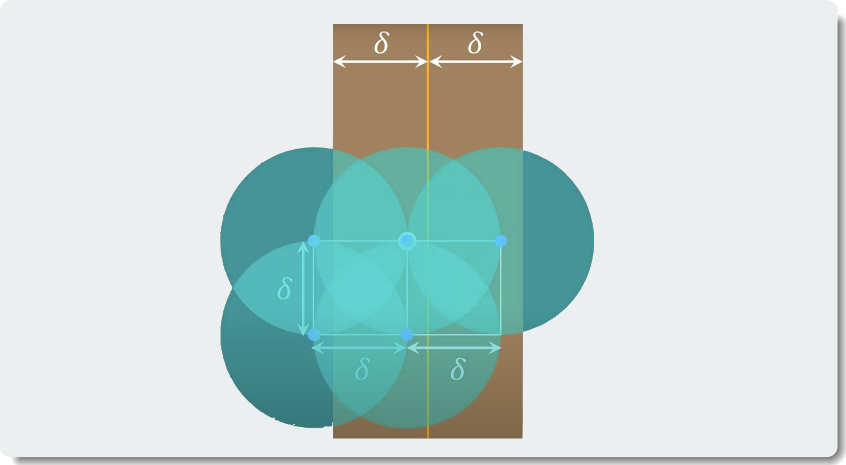
Is 15 elements optimal?

Geometry Fun Question



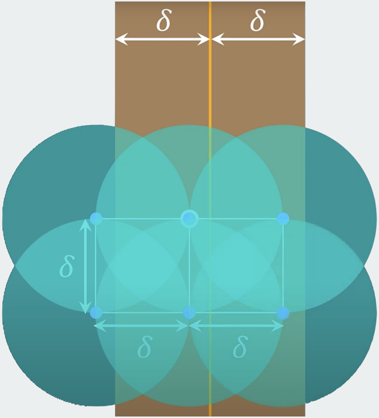
Is 15 elements optimal?

Geometry Fun Question



Is 15 elements optimal?

Geometry Fun Question



Completing the Analysis

Correctness of the Algorithm

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- 🤔 How many recursive calls?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- 🤔 What is the size of the recursive calls?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
  - $15 \cdot |S| = O(n)$.

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y $(O(n \log n))$.
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
  - $15 \cdot |S| = O(n)$.
- 🤔 What is the recurrence?

Completing the Analysis

Correctness of the Algorithm

- By induction on the number of points.
- Use the definition of the algorithm and the claims establish in Step 3.

Runtime of the Algorithm

- Sorting by x and by y ($O(n \log n)$).
- How many recursive calls? 2.
- What is the size of the recursive calls? $n/2$.
- Work per call: check points in S.
  - $15 \cdot |S| = O(n)$.
- What is the recurrence?

$$T(n) \leq 2T(n/2) + cn = O(n \log n) \; .$$

Closest Pair Problem (Divide & Conquer)

```
def dist(p1, p2):
    return ((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)**0.5

def brute_force(P):
    return min([dist(P[i], P[j]) for i in range(len(P)) for j in range(i + 1, len(P))], default=float('inf'))

def closest_split_pair(Px, Py, delta, best_pair):
    middle = Px[len(Px) // 2][0]
    S = [p for p in Py if middle - delta <= p[0] <= middle + delta]
    best = delta
    for i in range(len(S) - 1):
        min_dist, j_best = min((dist(S[i], S[j]), j) for j in range(i + 1, min(i + 7, len(S))))
        best, best_pair = (min_dist, (i, j_best)) if min_dist <= best else (best, best_pair)
    return best, best_pair
```

Closest Pair Problem (Divide & Conquer)

```python
def closest_pair_rec(Px, Py):
    if len(Px) <= 3:
        return brute_force(Px)

    mid = len(Px) // 2
    Qx = Px[:mid]
    Rx = Px[mid:]

    midpoint = Px[mid][0]
    Qy = [point for point in Py if point[0] <= midpoint]
    Ry = [point for point in Py if point[0] > midpoint]

    (d1, pair1) = closest_pair_rec(Qx, Qy)
    (d2, pair2) = closest_pair_rec(Rx, Ry)
    d, best_pair = (d1, pair1) if d1 <= d2 else (d2, pair2)
    (d3, pair3) = closest_split_pair(Px, Py, d, best_pair)

    return (d, best_pair) if d <= d3 else (d3, pair3)

def closest_pair(points):
    Px = sorted(points, key=lambda x: x[0])
    Py = sorted(points, key=lambda x: x[1])
    return closest_pair_rec(Px, Py)
```

Max Subarray

Max Subarray

Problem

Given an array A of integers, find the (non-empty) contiguous subarray of A of maximum sum.

Max Subarray

Problem

Given an array A of integers, find the (non-empty) contiguous subarray of A of maximum sum.

Exercise – Teams of 3 or so

- Solve the problem in $\Theta(n^2)$.
- Solve the problem in $O(n \log n)$.
- Prove correctness and complexity.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarrays

Input : Array A of n ints.
Output: Max subarray in A.

```
Let M be an empty array
for i := 1 to len(A) do
    for j := i to len(A) do
        if sum(A[i..j]) > sum(M) then
            M := A[i..j]
        end
    end
end
return M
```

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarra

Input : Array A of n ints.
Output: Max subarray in A.

```
Let M be an empty array
for i := 1 to len(A) do
    for j := i to len(A) do
        if sum(A[i..j]) > sum(M
            M := A[i..j]
        end
    end
end
return M
```

Analysis

- Correct: Checks all possible contiguous subarrays.

Part 1: Give a $\Theta(n^2)$ solution.

Algorithm: CheckAllSubarra

Input : Array A of n ints.
Output: Max subarray in A.
Let M be an empty array
for $i := 1$ to $len(A)$ **do**
    for $j := i$ to $len(A)$ **do**
        if $sum(A[i..j]) > sum(M$
            $M := A[i..j]$
        end
    end
end
return M

Analysis

- Correct: Checks all possible contiguous subarrays.
- Complexity:
    - Re-calculating the sum will make it $O(n^3)$. Key is to calculate the sum as you iterate.
    - For each i, check $n - i + 1$ ends. Overall:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2) .$$

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.
if $|A| = 1$ **then return** $A[1]$
A_1 := MaxSubarray(Front-half of A)
A_2 := MaxSubarray(Back-half of A)
M :=MidMaxSubarray(A)
return Array with max sum of $\{A_1, A_2, M\}$

PART 2: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: MAXSUBARRAY

Input : Array A of n ints.
Output: Max subarray in A.

if $|A| = 1$ **then return** $A[1]$
A_1 := MAXSUBARRAY(Front-half of A)
A_2 := MAXSUBARRAY(Back-half of A)
M :=MIDMAXSUBARRAY(A)
return Array with max sum of $\{A_1, A_2, M\}$

Algorithm: MIDMAXSUBARRAY

Input : Array A of n ints.
Output: Max subarray that crosses midpoint A.

m := mid-point of A
L := max subarray in $A[i, m-1]$ for $i = m-1 \to 1$
R := max subarray in $A[m, j]$ for $j = m \to n$
return $L \cup R$ `// subarray formed by combining` L `and` R.

Part 2: Give an $O(n \log n)$ solution.

Algorithm: MaxSubarray

Input : Array A of n ints.
Output: Max subarray in A.
if $|A| = 1$ **then return** $A[1]$
A_1 := MaxSubarray(Front-half of A)
A_2 := MaxSubarray(Back-half of A)
M :=MidMaxSubarray(A)
return Array with max sum of $\{A_1, A_2, M\}$

Analysis

- Correctness: By induction, A_1 and A_2 are max for subarray and M is max mid-crossing array.
- Complexity: Same recurrence as MergeSort.

Appendix

References

Image Sources I

https://brand.wisc.edu/web/logos/

https://people.csail.mit.edu/virgi/