Divide & Conquer algorithms

- **Strategy:**
  - Divide the problem into smaller subproblems of the same type of problem
  - Solve the subproblems recursively
  - Combine the answers to solve the original problem

- **The work is done in three places:**
  - In partitioning the problem into subproblems
  - In solving the basic cases at the tail of the recursion
  - In merging the answers of the subproblems to obtain the solution of the original problem

---

Conventional product of polynomials

**Example:**

\[
P(x) = 2x^3 + x^2 - 4
\]

\[
Q(x) = x^2 - 2x + 3
\]

\[
(P \cdot Q)(x) = 2x^5 + (-4 + 1)x^4 + (6 - 2)x^3 + 8x - 12
\]

\[
(P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12
\]

---

Function `PolynomialProduct(P, Q)`

```plaintext
function PolynomialProduct(P, Q)
    // P and Q are vectors of coefficients.
    // Returns R = P \times Q.
    // degree(P) = size(P)-1, degree(Q) = size(Q)-1.
    // degree(R) = degree(P)+degree(Q).
    R = vector with size(P)+size(Q)-1 zeros;
    for each P_i
        for each Q_j
            R_{i+j} = R_{i+j} + P_i \cdot Q_j
    return R
```

---

**Complexity analysis:**

- Multiplication of polynomials of degree \( n \): \( O(n^2) \)
- Addition of polynomials of degree \( n \): \( O(n) \)
Product of polynomials: Divide & Conquer

Assume that we have two polynomials with \( n \) coefficients (degree \( n - 1 \))

\[
\begin{array}{c|c|c}
\text{P:} & n-1 & n/2 \ 0 \\
\hline
P_L & P_R \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{Q:} & \quad & \\
\hline
Q_L & Q_R \\
\end{array}
\]

\[ P(x) \cdot Q(x) = P_L(x) \cdot Q_L(x) \cdot x^n + (P_R(x) \cdot Q_L(x) + P_L(x) \cdot Q_R(x)) \cdot x^{n/2} + P_R(x) \cdot Q_R(x) \]

\[ T(n) = 4 \cdot T(n/2) + O(n) = O(n^2) \quad \Leftarrow \text{Shown later} \]

Product of complex numbers

- The product of two complex numbers requires four multiplications:
  \[
  (a + bi)(c + di) = ac - bd + (bc + ad)i
  \]

- Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: \( ac, bd \) and \( (a + b)(c + d) \)
  \[
  bc + ad = (a + b)(c + d) - ac - bd
  \]

- A similar observation applies for polynomial multiplication.

Product of polynomials with Gauss’s trick

\[
\begin{align*}
R_1 &= P_LQ_L \\
R_2 &= P_RQ_R \\
R_3 &= (P_L + P_R)(Q_L + Q_R)
\end{align*}
\]

\[
PQ = P_LQ_L x^n + (P_RQ_L + P_LQ_R) x^{n/2} + P_RQ_R
\]

\[
T(n) = 3T(n/2) + O(n)
\]

Polynomial multiplication: recursive step

\[
\begin{array}{c|c|c|c|c|c|c|c}
P & 1 & -2 & 3 & 2 & 0 & -1 \\
\hline
Q & 2 & 1 & 0 & -1 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_L & 1 & -2 & 3 & 2 & 0 & -1 \\
\hline
Q_L & 2 & 1 & 0 & -1 & 3 & 0 \\
\hline
P_L + P_R & 3 & -2 & 2 & 1 & 4 & 0 \\
\hline
P_R & 2 & 0 & -1 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c}
P_L & 1 & -2 & 3 & 2 & 0 & -1 \\
\hline
Q_R & -2 & 6 & 1 & -3 & 0 \\
\hline
Q_L + Q_R & 1 & 4 & 0 \\
\hline
Q_R & -1 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R + P_RQ_L & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c}
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
P_LQ_R & 3 & 7 & -11 & 8 & 0 \\
\hline
P_LQ_R + P_RQ_L & 2 & -3 & 4 & 3 & 0 \\
\hline
\end{array}
\]
**Pattern of recursive calls**

- **Branching factor:** 3

- **Levels:** \( \log_2 n \)

**Complexity analysis**

- The time spent at level \( k \) is

\[
3^k \cdot O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \cdot O(n)
\]

- For \( k = 0 \), runtime is \( O(n) \).

- For \( k = \log_2 n \), runtime is \( O\left(3^{\log_2 n}\right) \), which is equal to \( O\left(n^{\log_3 3}\right) \).

- The runtime per level increases geometrically by a factor of \( 3/2 \) per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series.

- Therefore, the complexity is \( O(n^{1.59}) \).

**A popular recursion tree**

- **Branching factor:** 2

- **Levels:** \( \log_2 n \)

**Examples**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Branch</th>
<th>c</th>
<th>Runtime equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power ((x^y))</td>
<td>1</td>
<td>0</td>
<td>(T(y) = T(y/2) + O(1))</td>
</tr>
<tr>
<td>Binary search</td>
<td>1</td>
<td>0</td>
<td>(T(n) = T(n/2) + O(1))</td>
</tr>
<tr>
<td>Merge sort</td>
<td>2</td>
<td>1</td>
<td>(T(n) = 2 \cdot T(n/2) + O(n))</td>
</tr>
<tr>
<td>Polynomial product</td>
<td>4</td>
<td>1</td>
<td>(T(n) = 4 \cdot T(n/2) + O(n))</td>
</tr>
<tr>
<td>Polynomial product (Gauss)</td>
<td>3</td>
<td>1</td>
<td>(T(n) = 3 \cdot T(n/2) + O(n))</td>
</tr>
</tbody>
</table>

Example: efficient sorting algorithms.

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

Algorithms may differ on the amount of work done at each level: \(O(n^c)\)
Master theorem

- Typical pattern for Divide & Conquer algorithms:
  - Split the problem into \( a \) subproblems of size \( n/b \)
  - Solve each subproblem recursively
  - Combine the answers in \( O(n^c) \) time

- Running time: \( T(n) = a \cdot T(n/b) + O(n^c) \)

- Master theorem:
  \[
  T(n) = \begin{cases} 
  O(n^c) & \text{if } c > \log_b a \quad (a < b^c) \\
  O(n^c \log n) & \text{if } c = \log_b a \quad (a = b^c) \\
  O(n^{\log_b a}) & \text{if } c < \log_b a \quad (a > b^c)
  \end{cases}
  \]

Master theorem: proof

- For simplicity, assume \( n \) is a power of \( b \).
- The base case is reached after \( \log_b n \) levels.
- The \( k \)th level of the tree has \( a^k \) subproblems of size \( n/b^k \).
- The total work done at level \( k \) is:
  \[
  a^k \times O\left(\frac{n}{b^k}\right)^c = O(n^c) \times \left(\frac{a}{b^c}\right)^k
  \]
- As \( k \) goes from 0 (the root) to \( \log_b n \) (the leaves), these numbers form a geometric series with ratio \( a/b^c \). We need to find the sum of such a series.

\[
T(n) = O(n^c) \cdot \left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \cdots + \frac{a^{\log_b n}}{b^{(\log_b n)c}}\right)
\]

Master theorem: recursion tree

- Case \( a/b^c < 1 \). Decreasing series. The sum is dominated by the first term \((k = 0): O(n^c)\).
- Case \( a/b^c < 1 \). Increasing series. The sum is dominated by the last term \((k = \log_b n):\)
  \[
  n^c \left(\frac{a}{b^c}\right)^{\log_b n} = n^c \left(\frac{a^{\log_b n}}{(b^{\log_b n})^c}\right) = a^{\log_b n} = \]
  \[
  = a^{(\log_a n)(\log_b a)} = n^{\log_b a}
  \]
- Case \( a/b^c = 1 \). We have \( O(\log n) \) terms all equal to \( O(n^c) \).
Running time: \[ T(n) = a \cdot T(n/b) + O(n^c) \]

\[ \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O(n \log_b a) & \text{if } a > b^c \end{cases} \]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>a</th>
<th>c</th>
<th>Runtime equation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power ((x^y))</td>
<td>1</td>
<td>0</td>
<td>(T(y) = T(y/2) + O(1))</td>
<td>(O(\log y))</td>
</tr>
<tr>
<td>Binary search</td>
<td>1</td>
<td>0</td>
<td>(T(n) = T(n/2) + O(1))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Merge sort</td>
<td>2</td>
<td>1</td>
<td>(T(n) = 2 \cdot T(n/2) + O(n))</td>
<td>(O(n \log n))</td>
</tr>
<tr>
<td>Polynomial product</td>
<td>4</td>
<td>1</td>
<td>(T(n) = 4 \cdot T(n/2) + O(n))</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>Polynomial product (Gauss)</td>
<td>3</td>
<td>1</td>
<td>(T(n) = 3 \cdot T(n/2) + O(n))</td>
<td>(O(n \log_2 3))</td>
</tr>
</tbody>
</table>

\(b = 2\) for all the examples

Quick sort with Hungarian, folk dance

Quick sort (Tony Hoare, 1959)

- Suppose that we know a number \(x\) such that one-half of the elements of a vector are greater than or equal to \(x\) and one-half of the elements are smaller than \(x\).
  - Partition the vector into two equal parts \((n − 1)\) comparisons.
  - Sort each part recursively.

- Problem: we do not know \(x\).

- The algorithm also works no matter which \(x\) we pick for the partition. We call this number the **pivot**.

- **Observation**: the partition may be unbalanced.

Quick sort: example

The key step of quick sort is the partitioning algorithm.

**Question**: how to find a good pivot?
Quick sort

Divide & Conquer © Dept. CS, UPC

Quick sort: partition

function Partition(A, left, right)
// A[left..right]: segment to be sorted
// Returns the middle of the partition with
//   A[middle] = pivot
//   A[left..middle-1] ≤ pivot
//   A[middle+1..right] > pivot

x = A[left]; // the pivot
i = left; j = right;

while i < j do
    while A[i] ≤ x and i ≤ right do i = i+1;
    while A[j] > x and j ≥ left do j = j-1;
    if i < j then swap(A[i], A[j]);

swap(A[left], A[j]);
return j;

Quick sort partition: example

pivot

6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14
6 2 4 5 10 9 12 1 15 7 3 13 8 11 16 14
6 2 4 5 3 9 12 1 15 7 10 13 8 11 16 14
6 2 4 5 3 1 12 9 15 7 10 13 8 11 16 14
1 2 4 5 3 6 12 9 15 7 10 13 8 11 16 14
1 2 4 5 3 6 12 9 15 7 10 13 8 11 16 14

middle

Quick sort: algorithm

function Qsort(A, left, right)
// A[left..right]: segment to be sorted

if left < right then
    mid = Partition(A, left, right);
    Qsort(A, left, mid-1);
    Qsort(A, mid+1, right);

Quick sort: algorithm

function Qsort(A, left, right)
// A[left..right]: segment to be sorted
if left < right then
    mid = Partition(A, left, right);
    Qsort(A, left, mid-1);
    Qsort(A, mid+1, right);
function HoarePartition(A, left, right)
// A[left..right]: segment to be sorted.
// Output: The left part has elements ≤ than the pivot.
// The right part has elements ≥ than the pivot.
// Returns the index of the last element of the left part.
x = A[left]; // the pivot
i = left-1; j = right+1;

while true do
  do i = i+1; while A[i] < x;
  do j = j-1; while A[j] > x;
  if i ≥ j then return j;
  swap(A[i], A[j]);

Divide & Conquer © Dept. CS, UPC

Quick sort partition: example
6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14

pivot
6 2 8 5 10 9 12 1 15 7 3 13 4 11 16 14

First swap: 4 is a sentinel for R; 6 is a sentinel for L → no need to check for boundaries
4 2 3 5 10 9 12 1 15 7 8 13 6 11 16 14

j (middle)
4 2 3 5 10 9 12 1 15 7 8 13 6 11 16 14
4 2 3 5 1 9 12 10 15 8 13 6 11 16 14
4 2 3 5 1 9 12 10 15 7 8 13 6 11 16 14

Quick sort with Hoare’s partition

function Qsort(A, left, right)
// A[left..right]: segment to be sorted
// K is a break-even size in which insertion sort is more efficient than quick sort.
if right - left ≥ K then
  mid = HoarePartition(A, left, right);
  Qsort(A, left, mid);
  Qsort(A, mid+1, right);

function Sort(A):
  Qsort(A, 0, A.size()-1);
  InsertionSort(A);
Quick sort: complexity analysis

- The partition algorithm is $O(n)$.
- Assume that the partition is balanced:
  $$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$
- Worst case runtime: the pivot is always the smallest element in the vector $\Rightarrow O(n^2)$
- Selecting a good pivot is essential. There are different strategies, e.g.,
  - Take the median of the first, last and middle elements
  - Take the pivot at random

The runtime if $x_i$ is selected as pivot is:
$$T(n) = n - 1 + T(i - 1) + T(n - i)$$

Full-history recurrence relation (proof)

A recurrence that depends on all the previous values of the function.

$$nT(n) = n(n - 1) + 2 \sum_{i=0}^{n-1} T(i)$$

$$(n + 1)T(n + 1) = (n + 1)n + 2 \sum_{i=0}^{n} T(i)$$

$$T(n + 1) = n + 2 \frac{n+1}{n+1} T(n) + \frac{2n}{n+1} \leq \frac{n + 2}{n + 1} T(n) + 2$$

$$T(n) \leq 2 + \frac{n + 1}{n} \left(2 + \frac{n}{n - 1} \left(2 + \frac{n - 1}{n - 2} \left(\frac{4}{3}\right)\right)\right)$$

$$T(n) \leq 2 \left(1 + \frac{n + 1}{n} + \frac{n + 1}{n - 1} + \frac{n + 1}{n - 2} + \cdots + \frac{n + 1}{n - n - 2} + \cdots + \frac{n + 1}{n - n - 2} \cdots \frac{4}{3}\right)$$

$$T(n) \leq 2(n + 1) \left(\frac{1}{n + 1} + \frac{1}{n - 1} + \frac{1}{n - 2} + \cdots + \frac{1}{3}\right) = 2(n + 1) \left(\frac{1}{n + 1} + \frac{1}{n - 1} + \cdots + \frac{1}{3}\right)$$

$$T(n) \leq 2(n + 1)(H(n + 1) - 1.5)$$
Quick sort: complexity analysis summary

• Runtime of quicksort:
  \[ T(n) = O(n^2) \]
  \[ T(n) = \Omega(n \log n) \]
  \[ T_{\text{avg}}(n) = O(n \log n) \]

• Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.

• Possible solution: use random pivots.

The selection problem

• Given a collection of \( N \) elements, find the \( k \)th smallest element.

• Options:
  – Sort a vector and select the \( k \)th location: \( O(N \log N) \)
  – Read \( k \) elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only \( k \) elements are maintained in the vector. Complexity: \( O(kN) \). Why?

The selection problem using a heap

• Algorithm:
  – Build a heap from the collection of elements: \( O(N) \)
  – Remove \( k \) elements: \( O(k \log N) \)
  – Note: heaps will be seen later in the course

• Complexity:
  – In general: \( O(N + k \log N) \)
  – For small values of \( k \), i.e., \( k = O(N/\log N) \), the complexity is \( O(N) \).
  – For large values of \( k \), the complexity is \( O(k \log N) \).

Quick sort with Hoare’s partition

```java
function Qsort(A, left, right)

// A[left..right]: segment to be sorted

if left < right then
  mid = HoarePartition(A, left, right);
  Qsort(A, left, mid);
  Qsort(A, mid+1, right);
```

Divide & Conquer © Dept. CS, UPC
Quick select with Hoare’s partition

// Returns the k-th smallest element in A
// Note: Assume that A=[A₁,A₂,...,Aₙ] and
// 1 ≤ k ≤ n. The smallest element has k = 1.

function Qselect(A, left, right, k)
    if left == right then return A[left];
    mid = HoarePartition(A, left, right);
    // We only need to sort one half of A
    if k ≤ mid then Qselect(A, left, mid, k);
    else Qselect(A, mid+1, right, k - mid);

The Closest-Points problem

• Input: A list of n points in the plane
  {(x₁, y₁), (x₂, y₂), ..., (xₙ, yₙ)}
• Output: The pair of closest points
• Simple approach: check all pairs → O(n²)
• We want an O(n log n) solution!

Quick Select: complexity

• Assume that the partition is balanced:
  – Quick sort: T(n) = 2T(n/2) + O(n) = O(n log n)
  – Quick select: T(n) = T(n/2) + O(n) = O(n)
• The average linear time complexity can be achieved by choosing good pivots (similar strategy and complexity computation to qsort).

The Closest-Points problem

• We can assume that the points are sorted by the x-coordinate. Sorting the points is free from the complexity standpoint (O(n log n)).
• Split the list into two halves. The closest points can be both at the left, both at the right or one at the left and the other at the right (center).
• The left and right pairs are easy to find (recursively). How about the pairs in the center?
The Closest-Points problem

- Let $\delta = \min(\delta_L, \delta_R)$. We only need to compute $\delta_C$ if it improves $\delta$.
- We can define a strip around the center with distance $\delta$ at the left and right. If $\delta_C$ improves $\delta$, then the points must be within the strip.
- In the worst case, all points can still reside in the strip.
- But how many points do we really have to consider?

The Closest-Points problem: algorithm

- Sort the points according to their $x$-coordinates.
- Divide the set into two equal-sized parts.
- Compute the min distance at each part (recursively). Let $\delta$ be the minimal of the two minimal distances.
- Eliminate points that are farther than $\delta$ from the separation line.
- Sort the remaining points according to their $y$-coordinates.
- Scan the remaining points in the $y$ order and compute the distances of each point to its 7 neighbors.
The Closest-Points problem: complexity

• Initial sort using $x$-coordinates: $O(n \log n)$. It comes for free.

• Divide and conquer:
  – Solve for each part recursively: $2T(n/2)$
  – Eliminate points farther than $\delta$: $O(n)$
  – Sort remaining points using $y$-coordinates: $O(n \log n)$
  – Scan the remaining points in $y$ order: $O(n)$

$$T(n) = 2T(n/2) + O(n) + O(n \log n) = O(n \log^2 n)$$

• Can we do it in $O(n \log n)$? Yes, we need to sort by $y$ in a smart way.

• Let $Y$ a vector with the points sorted by the $y$-coordinates. This can be done initially for free.

• Each time we partition the set of points by the $x$-coordinate, we also partition $Y$ into two sorted vectors (using an “unmerging” procedure with linear complexity)

```plaintext
Y_L = Y_R = \emptyset  // Initial lists of points
foreach p_i \in Y in ascending order of y do
  if p_i is at the left part then Y_L.push_back(p_i)
  else Y_R.push_back(p_i)
```

• Now, sorting the points by the $y$-coordinate at each iteration can be done in linear time, and the problem can be solved in $O(n \log n)$