Divide-and-Conquer

7  2  9  4 → 2  4  7  9

7  2  2  7
7 → 7 2 → 2

9  4  4  9
9 → 9 4 → 4
Divide-and-Conquer

Divide-and conquer is a general algorithm design paradigm:

- **Divide**: divide the input data $S$ in two or more disjoint subsets $S_1$, $S_2$, ...
- **Recur**: solve the subproblems recursively
- **Conquer**: combine the solutions for $S_1$, $S_2$, ..., into a solution for $S$

The base case for the recursion are subproblems of constant size

Analysis can be done using recurrence equations
Merge-Sort Review

Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:

- **Divide**: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
- **Recur**: recursively sort $S_1$ and $S_2$
- **Conquer**: merge $S_1$ and $S_2$ into a unique sorted sequence

**Algorithm mergeSort($S$)**

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted

```
if $S$.size() > 1
    $(S_1, S_2) \leftarrow \text{partition}(S, n/2)$
    mergeSort($S_1$)
    mergeSort($S_2$)
    $S \leftarrow \text{merge}(S_1, S_2)$
```
Merge-Sort

Algorithm `mergeSort(S)`

if `S.size()` > 1

    \((S_1, S_2) \leftarrow \text{partition}(S, n/2)\)
    `mergeSort(S_1)`
    `mergeSort(S_2)`
    \(S \leftarrow \text{merge}(S_1, S_2)\)

\(T(n) = \text{time for mergeSort on sequence } S \text{ of } n \text{ elements.}\)

\(O(1)\)  \(O(n)\)
\(O(n)\)  \(T(n/2)\)
\(O(n)\)  \(T(n/2)\)
\(O(n)\)

\[ T(n) = \begin{cases} 
O(1) & n = 1 \\
2T\left(\frac{n}{2}\right) + O(n) & n > 1 
\end{cases} \]

\(T(n) \in O(n \log n)\)
Median-Sort

Median-sort on an input sequence $S$ with $n$ elements consists of three steps:

- **Divide**: find the median of $S$ and partition $S$ into those elements less than the median ($S_1$) and those elements greater than or equal to the median ($S_2$).
- **Recur**: recursively sort $S_1$ and $S_2$
- **Conquer**: append $S_2$ to $S_1$

Algorithm $medianSort(S)$

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted

if $S.size() > 1$

$m \leftarrow median(S)$

$(S_1, S_2) \leftarrow mpartition(S, m)$

$medianSort(S_1)$

$medianSort(S_2)$

$S \leftarrow append(S_1, S_2)$
## Median-Sort

**Algorithm** \textit{medianSort}(S)

\begin{align*}
\text{if } S.\text{size}() &> 1 \\
& m \leftarrow \text{median}(S) \\
& (S_1, S_2) \leftarrow \text{partition}(S, m) \\
& \text{medianSort}(S_1) \\
& \text{medianSort}(S_2) \\
& S \leftarrow \text{append}(S_1, S_2)
\end{align*}

\[ T(n) = \text{time for mediansort on sequence } S \text{ of } n \text{ elements.} \]

\[ T(n) = \begin{cases} 
O(1) & n = 1 \\
2T\left(\frac{n}{2}\right) + O(n) & n > 1
\end{cases} \]

\[ T(n) \in O(n \log n) \]
Closest Pair (§33.4)

**Given:** A set $Q$ of points in two dimensions

**Find:** The distance between the closest pair of points of $Q$
Closest Pair – Main idea

**Divide:** the points into left half $Q_L$ and right half $Q_R$

**Recur:** solve the problem on both halves

**Conquer:** use the minimum from the two halves *and* the $Q_L$ to $Q_R$ pairs.
Let $\delta$ be the minimum of the distances returned from the recursive solutions to the $Q_L$ and $Q_R$ subproblems.

Consider only points within $\delta$ of the vertical dividing line.

Sort these points by y-coordinate, giving list $Y'$.

Check each point of $Y'$ against the **seven** points that follow it, computing their distance and updating a minimum distance $\delta'$ between all $Q_L$ to $Q_R$ pairs.
Why does 7 work?

All points in $Q_L$ are distance at least $\delta$ apart, as are all points in $Q_R$. In the extreme, a point $q$ of $Q_L$ can fit in three more points of $Q_L$ of decreasing $y$ before going a distance of $\delta$ in $y$-coordinate alone (a, b, and c as shown). It can fit in four points of $Q_R$ after it before passing $\delta$ in $y$-coordinate difference (q, d, e, and c as shown). Thus, any $Q_L$ to $Q_R$ pair with distance less than $\delta$ can be found by examining each point in the $2\delta$-wide strip around the dividing line and the seven points that follow it.
Algorithm ClosestPair(Q)

**Input:** A set of points Q

**Output:** The distance between the closest pair in Q

if |Q| > 3 then
    divide Q into left and right halves Q_L and Q_R
    \(d_L = \text{ClosestPair}(Q_L)\)
    \(d_R = \text{ClosestPair}(Q_R)\)
    \(\delta = \min(d_L, d_R)\)
    Q’ = points of Q within \(\delta\) of the dividing line
    sort Q’ by y-coordinate
    check distance of each point of Q’ to next seven points,
    maintaining minimum distance in \(\delta\)
    return \(\delta\)
## Analysis

<table>
<thead>
<tr>
<th>Algorithm ClosestPair(Q)</th>
<th>T(n) = time for ClosestPair on a set Q of n points</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A set of points Q</td>
<td><strong>Output:</strong> The distance between the closest pair in Q</td>
</tr>
<tr>
<td><strong>if</strong></td>
<td><strong>then</strong></td>
</tr>
<tr>
<td></td>
<td><strong>divide Q into left and right halves Q_L and Q_R</strong></td>
</tr>
<tr>
<td></td>
<td><strong>d_L = ClosestPair(Q_L)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>d_R = ClosestPair(Q_R)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>δ = min(d_L, d_R)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Q’ = points of Q within δ of the dividing line</strong></td>
</tr>
<tr>
<td></td>
<td><strong>sort Q’ by y-coordinate</strong></td>
</tr>
<tr>
<td></td>
<td><strong>check distance of each point of Q’ to next seven points,</strong></td>
</tr>
<tr>
<td></td>
<td><strong>maintaining minimum distance in δ</strong></td>
</tr>
<tr>
<td></td>
<td><strong>return δ</strong></td>
</tr>
</tbody>
</table>
Analysis

\[ T(n) = \begin{cases} 
O(1) & n \leq 3 \\
2T(n/2) + O(n \log n) & n > 3 
\end{cases} \]

The Master Method (from text) does not apply!
So we plug-and-chug:
\[
T(n) = 2T(n/2) + cn \log n \\
= 2(2T(n/4) + c(n/2) \log (n/2)) + cn \log n \\
= 4T(n/4) + cn \log (n/2) + cn \log n \\
= 8T(n/8) + cn \log (n/4) + cn \log (n/2) + cn \log n \\
\vdots \\
= (n/2)T(2) + cn (\log 2 + \log 4 + \ldots + \log (n/2) + \log n) \\
= nk/2 + cn( 1 + 2 + 3 + \ldots + \log n) \\
= nk/2 + cn(\log n)((\log n) + 1)/2 \\
= O(n) + O(n \log^2 n) \in O(n \log^2 n)
\]
We can get the actual pair of closest points by keeping this information with $d_L$, $d_R$, and $\delta$ when these values are computed or returned.

We can do better than $O(n \log^2 n)$:
Note that we’re getting $O(n \log n)$ from sorting in two places: when we divide, and when we put together the points in the $2\delta$-wide strip in the conquer step. Instead of doing this, we can do two sorts of all points in the beginning, one by $x$-coordinate and one by $y$-coordinate. Then, when we divide or combine elements of $Q$, we divide these sorted lists as well. Details are in the text.
This leads to a recursion:

$$T(n) = \begin{cases} 
O(1) & n \leq 3 \\
2T(n/2) + O(n) & n > 3 
\end{cases}$$

which is $O(n \log n)$. 

Drawing a line on a screen

In graphics, a basic problem is to draw a line (segment) on a display screen.
Suppose we start with an n by n grid of pixels and a line segment S. We ask: what pixels does S intersect?

There are several algorithms for answering this question.

We’ll consider the divide-and-conquer approach on the grid.

For convenience, consider n=2^k.
Line drawing

To draw S on the grid G, we will simply draw S on each quadrant of the grid. At a very high level:

\[
\text{DrawLine}(S, G) \\
\text{DrawLine}(S, 1^{st} \text{ quadrant of } G) \\
\text{DrawLine}(S, 2^{nd} \text{ quadrant of } G) \\
\text{DrawLine}(S, 3^{rd} \text{ quadrant of } G) \\
\text{DrawLine}(S, 4^{th} \text{ quadrant of } G)
\]

Analysis:

\(O(1)\) for figuring out quadrants
\(4T(n/2)\) for the subproblems

So \(T(n) = 4T(n/2) + c\), and \(T(1) = c\): thus \(T(n) \in O(n^2)\)
Line drawing

At a high level:

\[ \text{DrawLine}(S, G) \]

if(G is 1 by 1)

if(S intersects G)

setPixel(G)

else

figure out quadrants of G

DrawLine(S, 1^{st} quadrant of G)
DrawLine(S, 2^{nd} quadrant of G)
DrawLine(S, 3^{rd} quadrant of G)
DrawLine(S, 4^{th} quadrant of G)
Line drawing

We can save time if S doesn’t intersect the grid:

```
DrawLine(S, G)
    if(S does not intersect G)
        return
    else if(G is 1 by 1)
        setPixel(G)
    else
        figure out quadrants Q₁, Q₂, Q₃, Q₄ of G
        DrawLine(S, Q₁)
        DrawLine(S, Q₂)
        DrawLine(S, Q₃)
        DrawLine(S, Q₄)
```
Better Analysis

\[ \text{DrawLine}(S, G) \]

\[
\begin{align*}
\text{if}(S \text{ does not intersect } G) & \quad O(1) \\
\text{return} & \quad O(1) \\
\text{else if}(G \text{ is 1 by 1}) & \quad O(1) \\
\text{setPixel}(G) & \quad O(1) \\
\text{else} & \\
\text{figure out quadrants } Q_1, Q_2, Q_3, Q_4 \text{ of } G & \quad O(1) \\
\text{DrawLine}(S, Q_1) \\
\text{DrawLine}(S, Q_2) \\
\text{DrawLine}(S, Q_3) \\
\text{DrawLine}(S, Q_4) \\
\end{align*}
\]

\[ 3T(n/2) + O(1) \]

So \( T(n) = 3T(n/2) + O(1), \ T(1) = c: \text{ thus } T(n) \text{ is } O(n^{\log 3}) \approx O(n^{1.585}) \]
Lower level pseudocode

```plaintext
DrawLine(S, G, (x₀, y₀), (x₁, y₁))
    if(S does not intersect the box from (x₀, y₀) to (x₁, y₁))
        return
    else if(x₁ - x₀ ≤ 1 and y₁ - y₀ ≤ 1)
        setPixel(G, x₀, y₀)
    else
        xₘ = (x₀ + x₁) / 2
        yₘ = (y₀ + y₁) / 2
        DrawLine(S, G, (xₘ, yₘ), (x₁, y₁))
        DrawLine(S, G, (x₀, yₘ), (xₘ, y₁))
        DrawLine(S, G, (x₀, y₀), (xₘ, yₘ))
        DrawLine(S, G, (xₘ, y₀), (x₁, yₘ))
```

© 2020 Shermer
A note on line drawing and pseudocode

Divide-and-conquer on the grid does not do well for this problem. There are iterative algorithms which take $O(n)$ time.

I used the problem simply as an example of an approach to try, and to show how with a little refinement one may be able to get better asymptotic bounds. I also wanted to illustrate the difference between high-level and low-level pseudocode.

Knowing which level to write pseudocode at is a matter of judgement that you can develop over time. It mainly depends on your audience and what aspects of the solution you are trying to convey. For the purposes of this course, assume your audience doesn’t know much.