Divide & Conquer

Jordi Cortadella and Jordi Petit
Department of Computer Science
Divide-and-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:

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

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

```plaintext
function PolynomialProduct(P, Q)
    // P and Q are vectors of coefficients.
    // Returns R = P × 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 ∙ Q_j

    return R
```

Complexity analysis:
- Multiplication of polynomials of degree n: O(n²)
- Addition of polynomials of degree n: O(n)
Product of polynomials: Divide & Conquer

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

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

\[
T(n) = 4 \cdot T(n/2) + O(n) = O(n^2)
\]

← 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

\[ R_1 = P_L Q_L \]
\[ R_2 = P_R Q_R \]
\[ R_3 = (P_L + P_R)(Q_L + Q_R) \]

\[
PQ = P_L Q_L x^n + \underbrace{(P_R Q_L + P_L Q_R)}_{R_3 - R_1 - R_2} x^{n/2} + \underbrace{P_R Q_R}_{R_2}
\]

\[ T(n) = 3T(n/2) + O(n) \]
Polynomial multiplication: recursive step

Divide & Conquer

© Dept. CS, UPC

\[ P = \begin{array}{cccccc}
1 & -2 & 3 & 2 & 0 & -1 \\
Q = \begin{array}{cccc}
2 & 1 & 0 & -1 \\
3 & 0 & 
\end{array}
\end{array} \]

\[ P_L \begin{array}{cccc}
1 & -2 & 3 \\
Q_L = \begin{array}{cccc}
2 & 1 & 0 \\
2 & -3 & 4 & 3 & 0 
\end{array}
\end{array} \]

\[ P_L + P_R = \begin{array}{cccc}
3 & -2 & 2 \\
\end{array} \]

\[ Q_L + Q_R = \begin{array}{cccc}
1 & 4 & 0 \\
3 & 10 & -6 & 8 & 0 \\
-2 & 6 & 1 & -3 & 0 
\end{array} \]

\[ P_L Q_R + P_R Q_L = \begin{array}{cccc}
3 & 7 & -11 & 8 & 0 \\
\end{array} \]

\[ P_R \begin{array}{cccc}
2 & 0 & -1 \\
Q_R = \begin{array}{cccc}
-1 & 3 & 0 \\
1 & -2 & 6 & 1 & -3 & 0 
\end{array}
\end{array} \]

\[ -2 & 6 & 1 & -3 & 0 
\]

\[ 2 & -3 & 4 & 3 & 0 
\]

\[ 2 & -3 & 4 & 6 & 7 & -11 & 6 & 6 & 1 & -3 & 0 \]
Pattern of recursive calls

- **Branching factor**: 3

![Diagram showing recursive calls with branching factor 3 and log₂ n levels.](image)
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 $O\left(n^{\log_2 3}\right)$.

• 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

$log_2 n$ levels

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: $O(n^c)$
## 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>
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: recursion tree

- Size $n$
- Size $n/b$
- Size $n/b^2$
- Size 1

Branching factor $a$

Depth $\log_b n$

Width $a^{\log_b n} = n^{\log_b a}$
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: proof

- **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 $O(n^c)$. 
Master theorem: examples

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

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

<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 (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 with Hungarian, folk dance
The key step of quick sort is the partitioning algorithm.

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

https://en.wikipedia.org/wiki/Quicksort
Quick sort: partition

```python
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;
```

Divide & Conquer

© Dept. CS, UPC

22
Quick sort partition: example

pivot

<table>
<thead>
<tr>
<th>6</th>
<th>2</th>
<th>8</th>
<th>5</th>
<th>10</th>
<th>9</th>
<th>12</th>
<th>1</th>
<th>15</th>
<th>7</th>
<th>3</th>
<th>13</th>
<th>4</th>
<th>11</th>
<th>16</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>15</td>
<td>7</td>
<td>3</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>15</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>12</td>
<td>9</td>
<td>15</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>9</td>
<td>15</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>16</td>
<td>14</td>
</tr>
</tbody>
</table>

middle
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]);

Admire a unique piece of art by Hoare: The first swap creates two sentinels. After that, the algorithm flies ...
Quick sort partition: example

pivot

First swap: 4 is a sentinel for R; 6 is a sentinel for L \(\rightarrow\) no need to check for boundaries

\[\begin{array}{cccccccccccccccc}
4 & 2 & 3 & 5 & 10 & 9 & 12 & 1 & 15 & 7 & 8 & 13 & 6 & 11 & 16 & 14 \\
\end{array}\]
Quick sort with Hoare’s partition

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);
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
Quick sort: complexity analysis

• Let us assume that $x_i$ is the $i$th smallest element in the vector.

• Let us assume that each element has the same probability of being selected as pivot.

• The runtime if $x_i$ is selected as pivot is:

$$T(n) = n - 1 + T(i - 1) + T(n - i)$$
Quick sort: complexity analysis

\[ T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) \]

\[ T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^{n} T(i - 1) + \frac{1}{n} \sum_{i=1}^{n} T(n - i) \]

\[ T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \leq 2(n + 1)(H(n + 1) - 1.5) \]

\[ H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n \] is the Harmonic series, that has a simple approximation: \( H(n) = \ln n + \gamma + O(1/n) \).

\( \gamma = 0.577 \ldots \) is Euler’s constant.

\[ T(n) \leq 2(n + 1)(\ln n + \gamma - 1.5) + O(1) = O(n \log n) \]
Quick sort: complexity analysis summary

- Runtime of quicksort:

\[ T(n) = \Theta(n^2) \]
\[ T(n) = \Omega(n \log n) \]
\[ T_{\text{avg}}(n) = \Theta(n \log n) \]

- 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 $O(k \log N)$. 
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);
Quick select with Hoare’s partition

// Returns the element at location k assuming
// A[left..right] would be sorted in ascending order.
// Pre: left ≤ k ≤ right.
// Post: The elements of A have changed their locations.

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);
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

- **Input:** A list of $n$ points in the plane 
  $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs $\rightarrow O(n^2)$
- We want an $O(n \log n)$ solution!

![Diagram of points with closest pair highlighted]
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?

![Diagram showing the closest points problem with points and lines representing the distances $\delta_L$, $\delta_C$, and $\delta_R$.]
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

Let us take all points in the strip and sort them by the \( y \)-coordinate. We only need to consider pairs of points with distance smaller than \( \delta \).

Once we find a pair \((p_i, p_j)\) with \( y \)-coordinates that differ by more than \( \delta \), we can move to the next \( p_i \).

```c
for (i=0; i < NumPointsInStrip; ++i)
    for (j=i+1; j < NumPointsInStrip; ++j)
        if (\( p_i \) and \( p_j \)'s \( y \)-coordinate differ by more than \( \delta \)) break; // Go to next \( p_i \)
        if (dist(\( p_i, p_j \)) < \( \delta \)) \( \delta = \) dist(\( p_i, p_j \));
```

But, how many pairs \((p_i, p_j)\) do we need to consider?
The Closest-Points problem

• For every point $p_i$ at one side of the strip, we only need to consider points from $p_{i+1}$.

• The relevant points only reside in the $2\delta \times \delta$ rectangle below point $p_i$. There can only be 8 points at most in this rectangle (4 at the left and 4 at the right). Some points may have the same coordinates.
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.
The Closest-Points problem: complexity

• 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)

\[
Y_L = Y_R = \emptyset \quad // \text{Initial lists of points}
\]

**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 $O(n \log n)$
A recurrence that depends on all the previous values of the function.

\[ T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \]

\[ nT(n) = n(n - 1) + 2 \sum_{i=0}^{n-1} T(i), \quad (n+1)T(n+1) = (n+1)n + 2 \sum_{i=0}^{n} T(i) \]

\[ (n + 1)T(n + 1) - nT(n) = (n + 1)n - n(n - 1) + 2T(n) = 2n + 2T(n) \]

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

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

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

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

\[ T(n) \leq 2(n + 1)(H(n + 1) - 1.5) = \Theta(n \log n) \]
Harmonic series

\[ H(n) = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \]

\[ \sum_{k=2}^{n+1} \frac{1}{k} \leq \int_{1}^{n+1} \frac{dx}{x} \leq \sum_{k=1}^{n} \frac{1}{k} \leq \ln(n+1) \]

\[ H(n) = \Theta(\log n) \]