Divide and Conquer Algorithms

T. M. Murali

February 18 and 20, 2008
Study three divide and conquer algorithms:
- Counting inversions.
- Finding the closest pair of points.
- Integer multiplication.

First two problems use clever conquer strategies.
Third problem uses a clever divide strategy.
Motivation

- Collaborative filtering: match one user’s preferences to those of other users.
- Meta-search engines: merge results of multiple search engines to into a better search result.

Fundamental question: how do we compare a pair of rankings? Suggestion: two rankings are very similar if they have few inversions.

Assume one ranking is the ordered list of integers from 1 to \( n \).

The other ranking is a permutation \( a_1, a_2, \ldots, a_n \) of the integers from 1 to \( n \).

The second ranking has an inversion if there exist \( i, j \) such that \( i < j \) but \( a_i > a_j \).

The number of inversions \( s \) is a measure of the difference between the rankings.

Question also arises in statistics: Kendall’s rank correlation of two lists of numbers is \( 1 - \frac{2s}{n(n-1)} \).
Motivation

- Collaborative filtering: match one user’s preferences to those of other users.
- Meta-search engines: merge results of multiple search engines to into a better search result.
- Fundamental question: how do we compare a pair of rankings?
Motivation

- Collaborative filtering: match one user’s preferences to those of other users.
- Meta-search engines: merge results of multiple search engines to into a better search result.
- Fundamental question: how do we compare a pair of rankings?
- Suggestion: two rankings are very similar if they have few inversions.
Motivation

▶ Collaborative filtering: match one user’s preferences to those of other users.
▶ Meta-search engines: merge results of multiple search engines to into a better search result.
▶ Fundamental question: how do we compare a pair of rankings?
▶ Suggestion: two rankings are very similar if they have few inversions.
  ▶ Assume one ranking is the ordered list of integers from 1 to $n$.
  ▶ The other ranking is a permutation $a_1, a_2, \ldots, a_n$ of the integers from 1 to $n$.
  ▶ The second ranking has an inversion if there exist $i, j$ such that $i < j$ but $a_i > a_j$.
  ▶ The number of inversions $s$ is a measure of the difference between the rankings.
▶ Question also arises in statistics: *Kendall’s rank correlation* of two lists of numbers is $1 - 2s/(n(n - 1))$. 
Counting Inversions

**Count Inversions**

**INSTANCE:** A list \( L = x_1, x_2, \ldots, x_n \) of distinct integers between 1 and \( n \).

**SOLUTION:** The number of pairs \((i, j), 1 \leq i < j \leq n\) such \( a_i > a_j \).
Counting Inversions

**Count Inversions**

**INSTANCE:** A list \( L = x_1, x_2, \ldots, x_n \) of distinct integers between 1 and \( n \).

**SOLUTION:** The number of pairs \((i, j), 1 \leq i < j \leq n\) such \( a_i > a_j \).

---

2 4 1 3 5

1 2 3 4 5

**Figure 5.4** Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.
Counting Inversions: Algorithm

- How many inversions can be there in a list of \( n \) numbers?
Counting Inversions: Algorithm

- How many inversions can be there in a list of $n$ numbers? $\Omega(n^2)$. We cannot afford to compute each inversion explicitly.
How many inversions can be there in a list of \( n \) numbers? \( \Omega(n^2) \). We cannot afford to compute each inversion explicitly.

Sorting removes all inversions in \( O(n \log n) \) time. Can we modify the Mergesort algorithm to count inversions?
Counting Inversions: Algorithm

How many inversions can be there in a list of \( n \) numbers? \( \Omega(n^2) \). We cannot afford to compute each inversion explicitly.

Sorting removes all inversions in \( O(n \log n) \) time. Can we modify the Mergesort algorithm to count inversions?

Candidate algorithm:
1. Partition \( L \) into two lists \( A \) and \( B \) of size \( n/2 \) each.
2. Recursively count the number of inversions in \( A \).
3. Recursively count the number of inversions in \( B \).
4. Count the number of inversions involving one element in \( A \) and one element in \( B \).
Counting Inversions: Conquer Step

Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs $a_i$ and $b_j$ such $a_i > b_j$. 

Key idea: problem is much easier if $A$ and $B$ are sorted!

Merge-and-Count procedure:
- Maintain a current pointer for each list.
- Maintain a variable count initialised to 0.
- Initialise each pointer to the front of the list.
- While both lists are nonempty:
  - Let $a_i$ and $b_j$ be the elements pointed to by the current pointers.
  - Append the smaller of the two to the output list.
  - If $b_j$ is the smaller, increment count by the number of elements remaining in $A$.
  - Advance the current pointer in the list that the smaller element belonged to.
- EndWhile
- Append the rest of the non-empty list to the output.
- Return count and the merged list.

Running time of this algorithm is $O(m)$. 

Counting Inversions: Conquer Step

- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs $a_i$ and $b_j$ such $a_i > b_j$.
- Key idea: problem is much easier if $A$ and $B$ are sorted!
Counting Inversions: Conquer Step

- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_m$, compute the number of pairs $a_i$ and $b_j$ such $a_i > b_j$.
- Key idea: problem is much easier if $A$ and $B$ are sorted!
- **Merge-and-Count** procedure:
  - Maintain a *current* pointer for each list.
  - Maintain a variable *count* initialised to 0.
  - Initialise each pointer to the front of the list.
  - While both lists are nonempty:
    - Let $a_i$ and $b_j$ be the elements pointed to by the *current* pointers.
    - Append the smaller of the two to the output list.
    - If $b_j$ is the smaller, increment *count* by the number of elements remaining in $A$.
    - Advance the current pointer in the list that the smaller element belonged to.
  - EndWhile
  - Append the rest of the non-empty list to the output.
  - Return *count* and the merged list.

Running time of this algorithm is $O(m)$.  

T. M. Murali February 18 and 20, 2008
Divide and Conquer Algorithms
Counting Inversions: Conquer Step

- Given lists $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots b_m$, compute the number of pairs $a_i$ and $b_j$ such $a_i > b_j$.
- Key idea: problem is much easier if $A$ and $B$ are sorted!
- **Merge-and-Count** procedure:
  - Maintain a current pointer for each list.
  - Maintain a variable count initialised to 0.
  - Initialise each pointer to the front of the list.
  - While both lists are nonempty:
    - Let $a_i$ and $b_j$ be the elements pointed to by the current pointers.
    - Append the smaller of the two to the output list.
    - If $b_j$ is the smaller, increment count by the number of elements remaining in $A$.
    - Advance the current pointer in the list that the smaller element belonged to.
  - EndWhile
  - Append the rest of the non-empty list to the output.
  - Return count and the merged list.
- Running time of this algorithm is $O(m)$. 
Counting Inversions: Final Algorithm

Sort-and-Count(L)

If the list has one element then
there are no inversions

Else

Divide the list into two halves:
   A contains the first \([n/2]\) elements
   B contains the remaining \([n/2]\) elements

\((r_A, A) = \text{Sort-and-Count}(A)\)
\((r_B, B) = \text{Sort-and-Count}(B)\)
\((r, L) = \text{Merge-and-Count}(A, B)\)

Endif

Return \(r = r_A + r_B + r\), and the sorted list \(L\)
Counting Inversions: Final Algorithm

Sort-and-Count(L)

If the list has one element then
there are no inversions
Else
Divide the list into two halves:

A contains the first \([n/2]\) elements
B contains the remaining \([n/2]\) elements

\((r_A, A) = \text{Sort-and-Count}(A)\)
\((r_B, B) = \text{Sort-and-Count}(B)\)
\((r, L) = \text{Merge-and-Count}(A, B)\)

Endif

Return \(r = r_A + r_B + r\), and the sorted list \(L\)

Running time \(T(n)\) of the algorithm is \(O(n \log n)\) because
\(T(n) \leq 2T(n/2) + O(n)\).
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, ... 
- Started in 1975 by Shamos and Hoey. 
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, ...
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, ldots.
- Started in 1975 by Shamos and Hoey.
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, ldots.

Closest Pair of Points

**INSTANCE:** A set \( P \) of \( n \) points in the plane

**SOLUTION:** The pair of points in \( P \) that are the closest to each other.
Computational Geometry

- Algorithms for geometric objects: points, lines, segments, triangles, spheres, polyhedra, \ldots.
- Started in 1975 by Shamos and Hoey.
- Problems studied have applications in a vast number of fields: ecology, molecular biology, statistics, computational finance, computer graphics, computer vision, \ldots.

Closest Pair of Points

**INSTANCE:** A set $P$ of $n$ points in the plane

**SOLUTION:** The pair of points in $P$ that are the closest to each other.

- At first glance, it seems any algorithm must take $\Omega(n^2)$ time.
- Shamos and Hoey figured out an ingenious $O(n \log n)$ divide and conquer algorithm.
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \).
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
Closest Pair: Set-up

- Let $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i = (x_i, y_i)$.
- Use $d(p_i, p_j)$ to denote the Euclidean distance between $p_i$ and $p_j$.
- Goal: find the pair of points $p_i$ and $p_j$ that minimise $d(p_i, p_j)$.
- How do we solve the problem in 1D?
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \).
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
- How do we solve the problem in 1D? Sort: closest pair must be adjacent in the sorted order.
Closest Pair: Set-up

- Let \( P = \{p_1, p_2, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \).
- Use \( d(p_i, p_j) \) to denote the Euclidean distance between \( p_i \) and \( p_j \).
- Goal: find the pair of points \( p_i \) and \( p_j \) that minimise \( d(p_i, p_j) \).
- How do we solve the problem in 1D? Sort: closest pair must be adjacent in the sorted order.
- The idea does not work in 2D.
Closest Pair: Algorithm Skeleton

1. Divide \( P \) into two sets \( Q \) and \( R \) of \( n/2 \) points such that each point in \( Q \) has \( x \)-coordinate less than any point in \( R \).
2. Recursively compute closest pair in \( Q \) and in \( R \), respectively.

▶ How do we implement this step in \( O(n) \) time?
Closest Pair: Algorithm Skeleton

1. Divide $P$ into two sets $Q$ and $R$ of $n/2$ points such that each point in $Q$ has $x$-coordinate less than any point in $R$.
2. Recursively compute closest pair in $Q$ and in $R$, respectively.
3. Let $\delta_1$ be the distance computed for $Q$, $\delta_2$ be the distance computed for $R$, and $\delta = \min(\delta_1, \delta_2)$.
4. Compute pair $(q, r)$ of points such that $q \in Q$, $r \in R$, $d(q, r) < \delta$ and $d(q, r)$ is the smallest possible.
Closest Pair: Algorithm Skeleton

1. Divide \( P \) into two sets \( Q \) and \( R \) of \( n/2 \) points such that each point in \( Q \) has \( x \)-coordinate less than any point in \( R \).
2. Recursively compute closest pair in \( Q \) and in \( R \), respectively.
3. Let \( \delta_1 \) be the distance computed for \( Q \), \( \delta_2 \) be the distance computed for \( R \), and \( \delta = \min(\delta_1, \delta_2) \).
4. Compute pair \((q, r)\) of points such that \( q \in Q, r \in R, d(q, r) < \delta \) and \( d(q, r) \) is the smallest possible.
   - How do we implement this step in \( O(n) \) time?
**Closest Pair: Conquer Step**

- Line $L$ passes through right-most point in $Q$.

---

**Figure 5.6** The first level of recursion: The point set $P$ is divided evenly into $Q$ and $R$ by the line $L$, and the closest pair is found on each side recursively.
Closest Pair: Conquer Step

- Line $L$ passes through right-most point in $Q$.
- Claim: If there exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$, then $q$ and $r$ are both within distance $\delta$ of $L$.

**Figure 5.6** The first level of recursion: The point set $P$ is divided evenly into $Q$ and $R$ by the line $L$, and the closest pair is found on each side recursively.
Closest Pair: Conquer Step

Line $L$ passes through right-most point in $Q$.

Claim: If there exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$, then $q$ and $r$ are both within distance $\delta$ of $L$.

Let $S$ be the set of points within distance $\delta$ of $L$ and let $S_y$ denote these points sorted by increasing $y$-coordinate.

Claim: There exist $q \in Q$, $r \in R$ such that $d(q, r) < \delta$ if and only if there exist $s, s' \in S$ such that $d(s, s') < \delta$.
Closest Pair: Packing Argument

- Intuition: if there are “too many” points in $S$ that are closer than $\delta$ to each other, then there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.
Closest Pair: Packing Argument

Intuition: if there are “too many” points in $S$ that are closer than $\delta$ to each other, then there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$. 

Closest Pair: Packing Argument

- Intuition: if there are “too many” points in $S$ that are closer than $\delta$ to each other, then there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$.

- For a point $s \in S$, let $s_y$ denote its $y$-coordinate.

- Converse of the claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$. 
Counting Inversions

Closest Pair: Packing Argument

- Intuition: if there are “too many” points in $S$ that are closer than $\delta$ to each other, then there must be a pair in $Q$ or in $R$ that are less than $\delta$ apart.

- Claim: If there exist $s, s' \in S$ such that $d(s, s') < \delta$ then $s$ and $s'$ are at most 15 indices apart in $S_y$.

- For a point $s \in S$, let $s_y$ denote its $y$-coordinate.

- Converse of the claim: If there exist $s, s' \in S$ such that $s'$ appears 16 or more indices after $s$ in $S_y$, then $s'_y - s_y \geq \delta$.

- Idea behind the proof: pack the plane with squares, argue that each square contains at most one point.

Each box can contain at most one input point.

Figure 5.7 The portion of the plane close to the dividing line $L$, as analyzed in the proof of (5.10).
## Closest Pair: Final Algorithm

**Closest-Pair(P)**

Construct $P_L$ and $P_R$ \((O(n \log n)\) time)
\((p^*_{n'}, p^*_{n}) = \text{Closest-Pair-Rec}(P_L, P_R)\)

**Closest-Pair-Rec(P, P_R)**

If \(|P| \leq 3\) then
  find closest pair by measuring all pairwise distances
Endif

Construct $Q_L$, $Q_R$, $R_L$, $R_R$ \((O(n)\) time)
\((q^*_0, q^*_1) = \text{Closest-Pair-Rec}(Q_L, Q_R)\)
\((r^*_0, r^*_1) = \text{Closest-Pair-Rec}(R_L, R_R)\)

\[
\delta = \min(d(q^*_0, q^*_1), d(r^*_0, r^*_1))
\]
\[x^* = \text{maximum } x\text{-coordinate of a point in set } Q\]
\[L = \{(x, y) : x = x^*\}\]
\[S = \text{points in } P \text{ within distance } \delta \text{ of } L.\]

Construct $S_P$ \((O(n)\) time)
For each point $s \in S_P$, compute distance from $s$
  to each of next 15 points in $S_P$
  Let $s, s'$ be pair achieving minimum of these distances
  \((O(n)\) time)

If \(d(s, s') \leq \delta\) then
  Return \((s, s')\)
Else if \(d(q^*_0, q^*_1) < d(r^*_0, r^*_1)\) then
  Return \((q^*_0, q^*_1)\)
Else
  Return \((r^*_0, r^*_1)\)
Endif
Closest Pair: Final Algorithm

Closest-Pair(P)

Construct $P_x$ and $P_y$ (O(n log n) time)

$(p_0^*, p_1^*) = \text{Closest-Pair-Rec}(P_x, P_y)$

Closest-Pair-Rec($P_x$, $P_y$)

If $|P| \leq 3$ then

find closest pair by measuring all pairwise distances

Endif

Construct $Q_x$, $Q_y$, $R_x$, $R_y$ (O(n) time)

$(q_0^*, q_1^*) = \text{Closest-Pair-Rec}(Q_x, Q_y)$

$(r_0^*, r_1^*) = \text{Closest-Pair-Rec}(R_x, R_y)$

$\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$

$x^* = \text{maximum } x\text{-coordinate of a point in set } Q$

$L = \{(x, y) : x = x^*\}$
Closest Pair: Final Algorithm

Closest-Pair-Rec\((P_x, P_y)\)

If \(|P| \leq 3\) then

find closest pair by measuring all pairwise distances

Endif

Construct \(Q_x, Q_y, R_x, R_y\) \((O(n)\) time)\)

\((q_0^*, q_1^*) = \text{Closest-Pair-Rec}(Q_x, Q_y)\)

\((r_0^*, r_1^*) = \text{Closest-Pair-Rec}(R_x, R_y)\)

\[\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))\]

\(x^* = \text{maximum } x\text{-coordinate of a point in set } Q\)

\(L = \{(x, y) : x = x^*\}\)

\(S = \text{points in } P \text{ within distance } \delta \text{ of } L.\)

Construct \(S_y\) \((O(n)\) time)\)

For each point \(s \in S_y\), compute distance from \(s\)
Closest Pair: Final Algorithm

\[
d = \min(d(q_0^*, q_1^*), \ d(r_0^*, r_1^*))
\]

\[
x^* = \text{maximum } x\text{-coordinate of a point in set } Q
\]

\[
L = \{(x, y) : x = x^*\}
\]

\[
S = \text{points in } P \text{ within distance } \delta \text{ of } L.
\]

Construct \( S_y \) (\( O(n) \) time)

For each point \( s \in S_y \), compute distance from \( s \)

to each of next 15 points in \( S_y \)

Let \( s, s' \) be pair achieving minimum of these distances

(\( O(n) \) time)

If \( d(s, s') < \delta \) then

Return \( (s, s') \)

Else if \( d(q_0^*, q_1^*) < d(r_0^*, r_1^*) \) then

Return \( (q_0^*, q_1^*) \)

Else

Return \( (r_0^*, r_1^*) \)

Endif
**Integer Multiplication**

**MULTIPLY INTEGERS**

**INSTANCE:** Two $n$-digit binary integers $x$ and $y$

**SOLUTION:** The product $xy$
# Integer Multiplication

**Multiply Integers**

**INSTANCE:** Two $n$-digit binary integers $x$ and $y$

**SOLUTION:** The product $xy$

- Multiply two $n$-digit integers.
Integer Multiplication

Multiply Integers

INSTANCE: Two $n$-digit binary integers $x$ and $y$

SOLUTION: The product $xy$

- Multiply two $n$-digit integers.
- Result has at most $2n$ digits.
**Integer Multiplication**

**Multiply Integers**

**INSTANCE:** Two $n$-digit binary integers $x$ and $y$

**SOLUTION:** The product $xy$

- Multiply two $n$-digit integers.
- Result has at most $2n$ digits.
- Algorithm we learnt in school takes $O(n^2)$ operations. Size of the input is not 2 but $2^n$.

![Multiplication Example](image)

**Figure 5.8** The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.
**Integer Multiplication**

**Multiply Integers**

**INSTANCE:** Two $n$-digit binary integers $x$ and $y$

**SOLUTION:** The product $xy$

- Multiply two $n$-digit integers.
- Result has at most $2n$ digits.
- Algorithm we learnt in school takes $O(n^2)$ operations. Size of the input is not 2 but $2n$,

\[
\begin{array}{c}
1100 \\
\times 1101 \\
\hline
1100 \\
12 \\
\times 13 \\
\hline
36 \\
12 \\
\hline
156
\end{array}
\begin{array}{c}
1100 \\
0000 \\
1100 \\
1100 \\
\hline
10011100
\end{array}
\]

**Figure 5.8** The elementary-school algorithm for multiplying two integers, in (a) decimal and (b) binary representation.
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let $x$ be split into $x_0$ (lower-order bits) and $x_1$ (higher-order bits) and $y$ into $y_0$ (lower-order bits) and $y_1$ (higher-order bits).

$$xy =$$
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first \( n/2 \) bits and last \( n/2 \) bits.
- Let \( x \) be split into \( x_0 \) (lower-order bits) and \( x_1 \) (higher-order bits) and \( y \) into \( y_0 \) (lower-order bits) and \( y_1 \) (higher-order bits).

\[
xy = (x_12^{n/2} + x_0)(y_12^{n/2} + y_0) \\
= x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0.
\]
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let $x$ be split into $x_0$ (lower-order bits) and $x_1$ (higher-order bits) and $y$ into $y_0$ (lower-order bits) and $y_1$ (higher-order bits).

\[
xy = (x_12^{n/2} + x_0)(y_12^{n/2} + y_0) = x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0.
\]

- Each of $x_1, x_0, y_1, y_0$ has $n/2$ bits, so we can compute $x_1y_1, x_1y_0, x_0y_1,$ and $x_0y_0$ recursively, and merge the answers in $O(n)$ time.
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first \( n/2 \) bits and last \( n/2 \) bits.
- Let \( x \) be split into \( x_0 \) (lower-order bits) and \( x_1 \) (higher-order bits) and \( y \) into \( y_0 \) (lower-order bits) and \( y_1 \) (higher-order bits).

\[
xy = (x_12^{n/2} + x_0)(y_12^{n/2} + y_0) \\
= x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0.
\]

- Each of \( x_1, x_0, y_1, y_0 \) has \( n/2 \) bits, so we can compute \( x_1y_1, x_1y_0, x_0y_1, \) and \( x_0y_0 \) recursively, and merge the answers in \( O(n) \) time.
- What is the running time \( T(n) \)?
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let $x$ be split into $x_0$ (lower-order bits) and $x_1$ (higher-order bits) and $y$ into $y_0$ (lower-order bits) and $y_1$ (higher-order bits).

\[
xy = (x_12^{n/2} + x_0)(y_12^{n/2} + y_0) = x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0.
\]

- Each of $x_1, x_0, y_1, y_0$ has $n/2$ bits, so we can compute $x_1y_1, x_1y_0, x_0y_1, \text{ and } x_0y_0$ recursively, and merge the answers in $O(n)$ time.
- What is the running time $T(n)$?

\[
T(n) \leq 4T(n/2) + cn
\]
Divide-and-Conquer Algorithm

- Assume integers are binary.
- Let us use divide and conquer by splitting each number into first $n/2$ bits and last $n/2$ bits.
- Let $x$ be split into $x_0$ (lower-order bits) and $x_1$ (higher-order bits) and $y$ into $y_0$ (lower-order bits) and $y_1$ (higher-order bits).

$$xy = (x_12^{n/2} + x_0)(y_12^{n/2} + y_0)$$
$$= x_1y_12^n + (x_1y_0 + x_0y_1)2^{n/2} + x_0y_0.$$

- Each of $x_1, x_0, y_1, y_0$ has $n/2$ bits, so we can compute $x_1y_1, x_1y_0, x_0y_1,$ and $x_0y_0$ recursively, and merge the answers in $O(n)$ time.
- What is the running time $T(n)$?

$$T(n) \leq 4T(n/2) + cn$$
$$\leq O(n^2)$$
Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1y_0$ and $x_0y_1$ independently; we just need their sum.

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

- We have three sub-problems of size $n/2$.

What is the running time $T(n)$?

$$T(n) \leq 3T(n/2) + cn \leq O(n \log^3 2) = O(n^{1.59})$$
Four sub-problems lead to an $O(n^2)$ algorithm.

How can we reduce the number of sub-problems?

1. We do not need to compute $x_1y_0$ and $x_0y_1$ independently; we just need their sum.
2. $x_1y_1 + (x_1y_0 + x_0y_1) + x_0y_0 = (x_0 + x_1)(y_0 + y_1)$
3. Compute $x_1y_1$, $x_0y_0$ and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1y_0 + x_0y_1)$ by subtraction.
4. We have three sub-problems of size $n/2$.

What is the running time $T(n)$?
Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1y_0$ and $x_0y_1$ independently; we just need their sum.
  - $x_1y_1 + (x_1y_0 + x_0y_1) + x_0y_0 = (x_0 + x_1)(y_0 + y_1)$
  - Compute $x_1y_1$, $x_0y_0$ and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1y_0 + x_0y_1)$ by subtraction.
  - We have three sub-problems of size $n/2$.
- What is the running time $T(n)$?

$$T(n) \leq 3T(n/2) + cn$$
Improving the Algorithm

- Four sub-problems lead to an $O(n^2)$ algorithm.
- How can we reduce the number of sub-problems?
  - We do not need to compute $x_1y_0$ and $x_0y_1$ independently; we just need their sum.
  - $x_1y_1 + (x_1y_0 + x_0y_1) + x_0y_0 = (x_0 + x_1)(y_0 + y_1)$
  - Compute $x_1y_1$, $x_0y_0$ and $(x_0 + x_1)(y_0 + y_1)$ recursively and then compute $(x_1y_0 + x_0y_1)$ by subtraction.
  - We have three sub-problems of size $n/2$.

- What is the running time $T(n)$?

\[
T(n) \leq 3T(n/2) + cn \\
\leq O(n^{\log_2 3}) = O(n^{1.59})
\]
Final Algorithm

Recursive-Multiply(x,y):
  Write \( x = x_1 \cdot 2^{n/2} + x_0 \)
  \( y = y_1 \cdot 2^{n/2} + y_0 \)
  Compute \( x_1 + x_0 \) and \( y_1 + y_0 \)
  \( p = \text{Recursive-Multiply}(x_1 + x_0, \ y_1 + y_0) \)
  \( x_1y_1 = \text{Recursive-Multiply}(x_1, y_1) \)
  \( x_0y_0 = \text{Recursive-Multiply}(x_0, y_0) \)
  Return \( x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0 \)