Divide-and-Conquer

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Chapter 4: Divide-and-Conquer

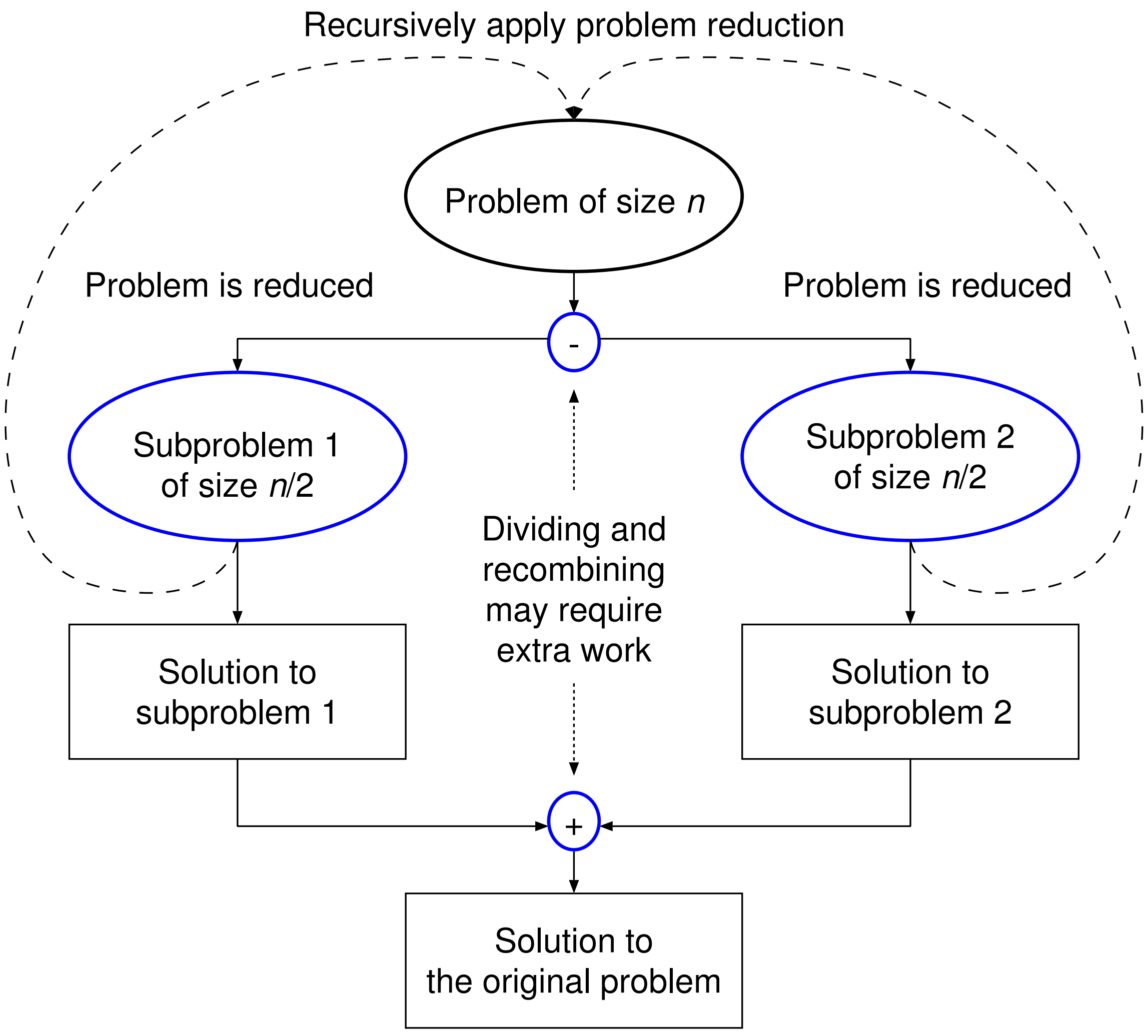
Topics: What we will cover today

- Mergesort
- Quicksort
- Binary search
- Binary tree traversals and related properties
- Large integer multiplication and Strassen's matrix multiplication
- Closest-pair problem and convex-hull problem (not discussed)

The Divide-and-Conquer Design Strategy (1)

1. A problem instance is divided into smaller instances of the same problem. Preferably the smaller instances have about the same input size.
2. The smaller instances are solved, usually either recursively again by divide-and-conquer or directly when the input size is small enough.
3. If necessary, the solutions of the smaller instances are combined to become a solution for the original problem.

The Divide-and-Conquer Design Strategy (2)



Notes on the Divide-and-Conquer Design Strategy

- In many typical cases a problem's instance of size n is divided into two sub-instances of size $n/2$.
- The divide-and-conquer method is well suited for parallel computers.

Recurrence for Analysis of Divide-and-Conquer Alg.

- Let a problem of input size n be divided into $a \geq 1$ subproblems of size n/b, $b \geq 1$. (We may assume n is a power of b.)
  - E.g., for binary search: $a = 2$ and $b = 2$.
- Let $f(n)$ be the cost of dividing the original problem of input size n into subproblems and for combining subsolutions into a solution.
- The recurrence is then

$$C(n) = aC\left(\frac{n}{b}\right) + f(n),$$

with some initial conditions.

Solving the Recurrence

- This recurrence can be solved by using the *Master Theorem*.
- Usually the function $C(n)$ is smooth and thus the asymptotic solution obtained when n is a power of b is also valid when n is not a power of b. (See text Appendix B for technical details.)

The Master Theorem

- THEOREM Let $C(n) = aC\left(\frac{n}{b}\right) + f(n)$. If $f(n) \in \Theta(n^d)$, $d \geq 0$, then

$$C(n) = \begin{cases} \Theta(n^d), & a < b^d; \\ \Theta(n^d \log n), & a = b^d; \\ \Theta(n^{\log_b a}), & a > b^d. \end{cases}$$

- Analogous results hold for the O and Ω notations.
- Examples:
  - Ex. 1: $C(n) = 4C(n/2) + n \Longrightarrow C(n) \in ?$
  - Ex. 2: $C(n) = 4C(n/2) + n^2 \Longrightarrow C(n) \in ?$
  - Ex. 3: $C(n) = 4C(n/2) + n^3 \Longrightarrow C(n) \in ?$

Use of the Master Theorem

- Ex. 1: $C(n) = 4C(n/2) + n$.

  $a = 4, b = 2, d = 1, a > b^d, C(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

- Ex. 2: $C(n) = 4C(n/2) + n^2$.

  $a = 4, b = 2, d = 2, a = b^d, C(n) \in \Theta(n^d \log n) = \Theta(n^2 \log n)$.

- Ex. 3: $C(n) = 4C(n/2) + n^3$.

  $a = 4, b = 2, d = 3, a < b^d, C(n) \in \Theta(n^d) = \Theta(n^3)$.

Mergesort

- Mergesort sorts the array $A[0..n-1]$ by first sorting the two subarrays $A[0..k-1]$ and $A[k..n-1]$, where $k = \lfloor n/2 \rfloor$.
- Note that subarray $A[0..k-1]$ has $\lfloor \frac{n}{2} \rfloor$ elements and subarray $A[k..n-1]$ has $\lceil \frac{n}{2} \rceil$ elements.
- The two sorted subarrays are then merged so that $A[0..n-1]$ is sorted.

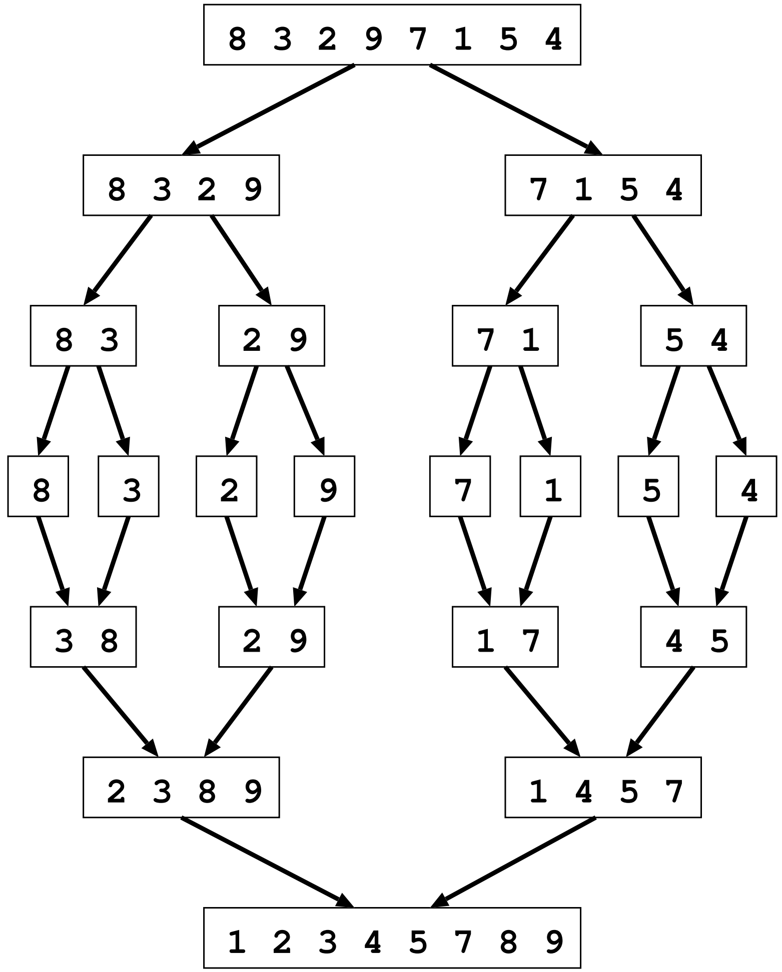
A Mergesort Algorithm

MERGESORT($A[0..n-1]$)
    // Input: An array $A[0..n-1]$ of orderable elements
    // Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$ then
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        MERGESORT($B[0..\lfloor n/2 \rfloor - 1]$)
        MERGESORT($C[0..\lceil n/2 \rceil - 1]$)
        MERGE(B, C, A)
    fi

A Mergesort Algorithm: Merge

MERGE($B[0..p-1], C[0..q-1], A[0..p+q-1]$)
    // Input: $B[0..p-1], C[0..q-1]$ are both sorted
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$ then
            $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
        else
            $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$ fi
        $k \leftarrow k + 1$ od
    if $i = p$ then
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else
        copy $B[i..p-1]$ to $A[k..p+q-1]$ fi

Mergesort Example



Comparison Counts for Merging: Lemma

- LEMMA The maximum number of comparisons for merging k_1 sorted keys and k_2 sorted keys is $2k_1 - 1$ when $k_1 = k_2$, and is $2k_1$ when $k_1 < k_2$.

Comparison Counts for Merging: Proof

- PROOF When $k_1 = k_2$ and merging $a_1 \leq \cdots \leq a_{k_1}$ and $b_1 \leq \cdots \leq b_{k_1}$, the maximum comparison count occurs when

$$b_1 \leq a_1 \leq b_2 \leq a_2 \leq \cdots \leq b_{k_1} \leq a_{k_1}.$$

- When $k_1 < k_2$ and merging $a_1 \leq \cdots \leq a_{k_1}$ and $b_1 \leq \cdots \leq b_{k_2}$, the maximum comparison count occurs when

$$b_1 \leq a_1 \leq b_2 \leq a_2 \leq \cdots \leq b_{k_1} \leq a_{k_1} \leq b_{k_1+1} \leq \cdots \leq b_{k_2}.$$

Analysis of Mergesort Algorithm

- We may assume that n is a power of 2. $n = k_1 + k_2$; $k_1 = k_2$.
- Let $C(n)$ be the worst case count of key comparisons.
- The recurrence is

$$C(n) = 2C(n/2) + n - 1, \quad n > 1, \quad C(1) = 0$$

- By the Master Theorem, we have

$$C(n) = \Theta(n \log n)$$

$$a = 2, b = 2, d = 1, a = b^d, C(n) \in \Theta(n^d \log n)$$

Mergesort: Summary

- Runtime, average case: $(n \log n)$
- Runtime, worst case: $(n \log n)$
- Space efficiency: $O(n)$
- Stability: yes

Quicksort

- Quicksort divides and conquers, but unlike mergesort that divides by array position, quicksort divides by array value.
- Quicksort partitions an array $A[0..n-1]$ into three parts:

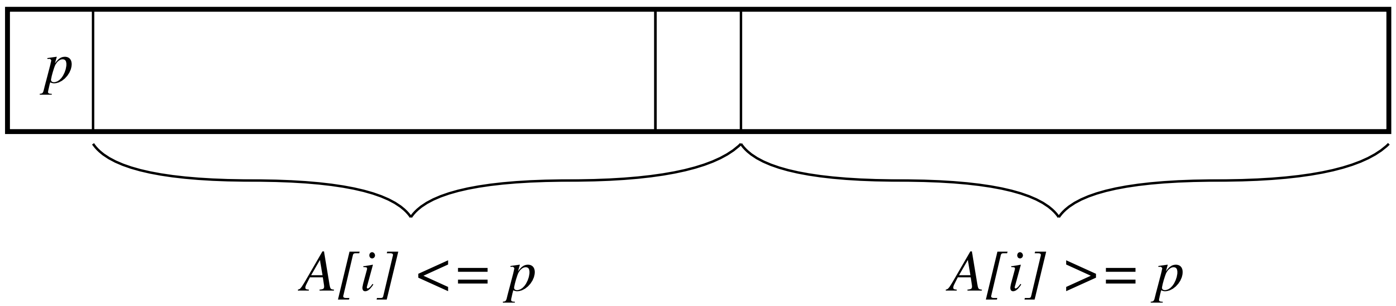
$$A[0..s-1] \leq A[s] \leq A[s+1..n-1].$$

$A[i..j] \leq X$ means $A[i], \cdots, A[j] \leq X$; $X \leq A[i..j]$ means $X \leq A[i], \cdots, A[j]$.

- That is, the partition puts the element $A[s]$ in its rightful position, elements $A[0..s-1]$ in their rightful positions relative to $A[s]$, and elements $A[s+1..n-1]$ in their rightful positions relative to $A[s]$.

A Quicksort Algorithm

- Select a pivot (partitioning element) – e.g., the first element.
- Rearrange the list so that all the elements in the first s positions are smaller than or equal to the pivot and all the elements in the remaining $n-s-1$ positions are larger than or equal to the pivot.



- Exchange the pivot with the last element in the first (i.e., $\leq$) subarray — the pivot is now in its final position.
- Sort the two subarrays recursively.

A Quicksort Algorithm

QUICKSORT($A[l..r]$)
// Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left
// and right indices l and r.
// Output: Subarray $A[l..r]$ sorted in nondecreasing order.
if $l < r$ then
    $s \leftarrow$ PARTITION($A[l..r]$) // s is split position
    QUICKSORT($A[0..s-1]$)
    QUICKSORT($A[s+1..r]$)
fi

A Partition Algorithm

```
PARTITION( A[l...r] )
    // A[l] is pivot
    p ← A[l]; i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        swap( A[i], A[j] )
    until i ≥ j
    swap( A[i], A[j] )                 // undo last swap when i ≥ j
    swap( A[l], A[j] )
    return j
```

Quicksort Analysis: Example

- For a Quicksort animation see:

`http://pages.stern.nyu.edu/%7epanos/java/Quicksort/`

Quicksort Analysis: Best Case

- The best case occurs when all the partitions divide their subarrays about evenly.
- Thus, the best case number of key comparisons is given by the recurrence

$$C(n) = 2C(n/2) + \Theta(n)$$

and we have $C(n) \in \Theta(n \log n)$ by the Master Theorem.

Quicksort Analysis: Worst Case

- The worst case occurs when all the partitions divide their subarrays into an empty array and an array with one fewer element than the given subarray.
- This happens when the array is already sorted.
- The worst case key comparison count is

$$C(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Quicksort Analysis: Average Case

- To find the average case key comparison count $C(n)$, we assume the pivot position s is equally likely to be any among $0..n-1$.
- The average case recurrence is

$$C(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C(s) + C(n-1-s)], \qquad n > 1,$$

$$C(0) = 0, C(1) = 0.$$

The Solution: $C(n) \approx 2n \ln n \approx 1.38 n \log_2 n.$

- Thus, only 38% more comparisons than in the best case.

Quicksort

- Improvements:
  - Better pivot selection: median of three partitioning.
  - Switch to insertion sort on small subfiles.
  - Elimination of recursion.
- These combine to 20%-25% improvement.

- Quicksort has good average case performance, but worst case is $\Theta(n^2)$. How can we avoid the worst case (i.e., an already sorted array)?

Quicksort: Summary

- Runtime, average case: $(n \log n)$
- Runtime, worst case: (n^2)
- Space efficiency: naive $O(n)$; $O(\log n)$
- Stability: no

Binary Search

- Binary search is very efficient for searching a given key among an array of sorted keys.
- It divides and conquers but is degenerate because it needs to solve only one subproblem and thus need not combine subsolutions.
- It can be implemented iteratively instead of recursively.

A Binary Search Algorithm

```
BINARYSEARCH
    // Input: a sorted array A[0 ... n − 1], a key K
    l ← 0; r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] then return m
        elsif K < A[m] then r ← m − 1
        else l ← m + 1
        fi
    od
    return −1
```

Binary Search Example

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Value	3	14	27	31	39	42	55	70	74	81	85	93	98
Iteration 1	l						m						r
Iteration 2								l		m			r
Iteration 3								l,m	r				

An Analysis of the Binary Search Algorithm

- The size of the larger of the subarrays $A[0..m-1]$ and $A[m+1..n-1]$, $m = \lfloor (n-1)/2 \rfloor$, is

$$\left\lfloor \frac{n}{2} \right\rfloor.$$

- To simplify we assume three-way comparisons; that is, with only one comparison we know if it is $A[m] < K$, $A[m] = K$ or $A[m] > K$.
- The recurrence for the worst case 3-way key count is

$$C(n) = C(\lfloor n/2 \rfloor) + 1, \quad n > 1; \quad C(1) = 1.$$

The Solution

- The solution of the preceding recurrence is

$$C(n) = \lfloor \log_2 n \rfloor + 1, \qquad n \geq 1.$$

- This is very fast, e.g., $C(10^6) = 20$.
- Note: Initially assume $n = 2^k$, but then generalize.
- The claim can be proved by strong mathematical induction and the fact that for any positive integer n there is an integer l such that

$$2^l \leq n \leq 2^{l+1} - 1.$$

Binary Tree Traversals and Related Properties

- Recall that a binary tree is a rooted tree such that every vertex has either no children or a left child or a right child or both.
- A binary tree can also be defined recursively as being either an empty tree having a left binary subtree and a right binary subtree.
- Thus a binary tree has three parts: root, left binary subtree, right binary subtree.
- This natural partition of a binary tree makes it inherently amenable to divide and conquer algorithms.

Full Binary Trees

- A binary tree is full if every vertex has either no children or two children.
- Let i and l be the numbers of internal vertices and leaves, respectively.
- In a full binary tree, every vertex except the root shares a parent with another vertex, thus

$$\frac{i+l-1}{2} = i.$$

- That is,

$$l = i + 1.$$

- This is a very useful property of full binary trees.

Finding the Height of a Binary Tree by Div-and-Conq

```
HEIGHT( T )
    // Input: A binary tree T
    if
        T = ∅ then return −1
    else
        return(max( HEIGHT(T_L), HEIGHT(T_R) ) +1)
    fi
```

Number of Empty Tree Checkings

- Let $C(n)$ be the number of times for checking if a binary tree is empty.
- We may represent an empty binary tree as a NULL vertex.
- By attaching these NULL vertices, a binary tree becomes a full binary tree whose leaves are NULL vertices.
- Since the algorithm visits every subtree root regardless if the subtree is empty, we have

$$C(n) = n + x = n + (n+1) = 2n + 1$$

where x is the number of NULL vertices attached.

Number of Additions

- Let $N(T)$ be the number of vertices of a binary tree T.
- The recurrence for the addition count $A(n)$ of the algorithm is

$$A(N(T)) = A(N(T_L)) + A(N(T_R)) + 1; \qquad A(0) = 0.$$

- By observation that there is one addition for each vertex in the binary tree, we immediately have

$$A(n) = n.$$

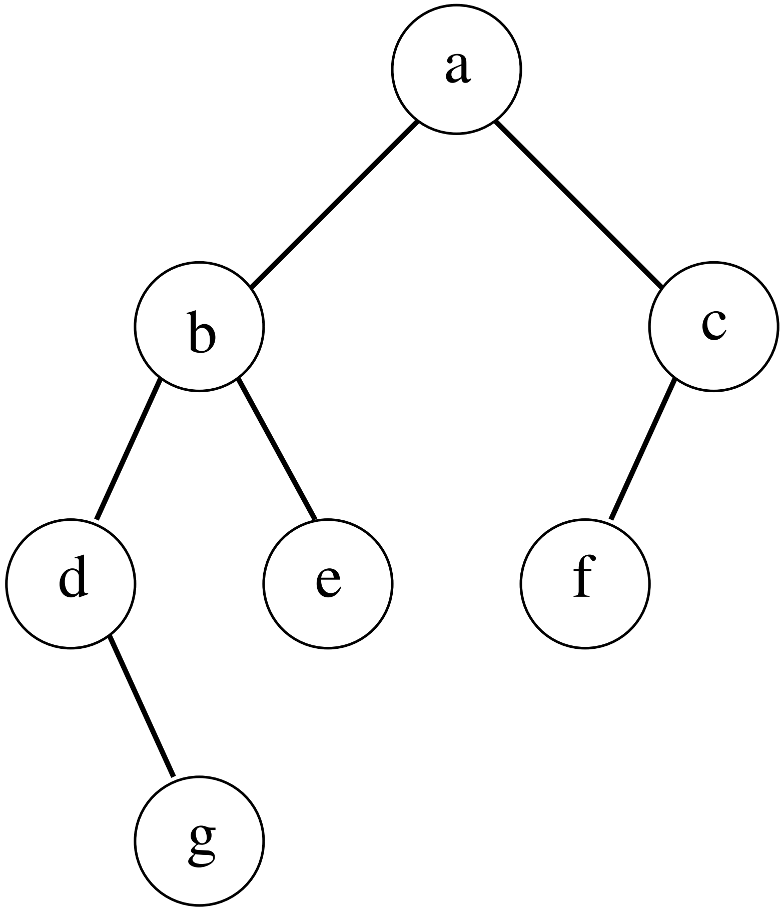
Preorder, Inorder, Postorder Traversal

```
PREORDER( T )
    if T ≠ ∅ then
        print T; PREORDER(T_L); PREORDER(T_R) fi
end

INORDER( T )
    if T ≠ ∅ then
        INORDER(T_L); print T; INORDER(T_R) fi
end

POSTORDER( T )
    if T ≠ ∅ then
        POSTORDER(T_L); POSTORDER(T_R); print T fi
end
```

Traversal Example



Preorder: a, b, d, g, e, c, f

Inorder: d, g, b, e, a, f, c

Postorder: g, d, e, b, f, c, a

Multiplication of Large Integers

- To multiply two n-digit numbers, n^2 digit multiplications are needed.
- Using divide and conquer, the number of digit multiplications can be reduced to $n^{1.585}$.
- For simplicity we assume n is a power of 2.
- Example:
  - Multiply 2 two-digit numbers, 23 and 14.
  - Representation: $23 = 2 \cdot 10^1 + 3 \cdot 10^0$ and $14 = 1 \cdot 10^1 + 4 \cdot 10^0$.
  - Multiplication: $23 \times 14 = (2 \cdot 10^1 + 3 \cdot 10^0) \times (1 \cdot 10^1 + 4 \cdot 10^0)$
    $= (2 \times 1)10^2 + (2 \times 4 + 3 \times 1)10^1 + (3 \times 4)10^0$.
  - Note: $2 \times 4 + 3 \times 1 = (2 + 3) \times (1 + 4) - 2 \times 1 - 3 \times 4$.

An Algorithm for the Multiplication of Large Integers

- An n-digit number x can be written as two $\frac{n}{2}$ numbers x_1 and x_0 as follows:

$$x = x_1 \times 10^{\frac{n}{2}} + x_0.$$

- To find the product c of two n-digit numbers a and b we compute

$$c = a \times b = (a_1 \times 10^{\frac{n}{2}} + a_0) \times (b_1 \times 10^{\frac{n}{2}} + b_0) = c_2 \times 10^n + c_1 \times 10^{\frac{n}{2}} + c_0$$

where $c_2 = a_1 b_1, c_0 = a_0 b_0$, and $c_1 = (a_1 + a_0)(b_1 + b_0) - c_2 - c_0$.

- Note: For $n = 2$ we reduced the number of multiplications from 4 to 3. However, we increased the number of additions!

A Remark

- The sum of two $\frac{n}{2}$-digit numbers $x_0 + x_1$ either remains a $\frac{n}{2}$-digit number or becomes a $\left(\frac{n}{2}+1\right)$-digit number with a leading digit 1.
- Thus we may write $x_1 + x_0 \times 10^{n/2} + x_2$ where x_3 is either 0 or 1 and x_2 is a $\frac{n}{2}$-digit number.
- The product $(x_3 \times 10^{n/2} + x_2)(y_3 \times 10^{n/2} + y_2) =$

  $x_3y_3 \times 10^n + (x_2y_3 + x_3y_2) \times 10^{n/2} + x_2y_2$.
- The product x_3y_3 and the sum $x_2y_3 + x_3y_2$ are trivial to compute because x_3, y_3 are either 0 or 1.

A Remark

- Consequently, the cost of multiplying two potentially $\left(\frac{n}{2}+1\right)$-digit numbers with 1 as the leading digit is the same as that of multiplying two $\frac{n}{2}$-digit numbers.

Analysis of the Integer Multiplication Algorithm

- Let $M(n)$ be the number of digit multiplications to multiply two n-digit numbers by the algorithm.
- Clearly the recurrence is

$$M(n) = 3M(n/2), \quad n > 1; \quad M(1) = 1.$$

- Backward substitution yields
$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] =$
$3^2M(2^{k-2}) = \cdots = 3^iM(2^{k-i}) = \cdots = 3^kM(2^{k-k}) = 3^k.$
- If $n = 2^k$, then

$$M(n) = 3^k = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

Strassen's Matrix Multiplication

- The product $C = AB$ of two order n square matrices A and B can be found with n^3 scalar multiplications:

  for $i \leftarrow 1$ to n do for $j \leftarrow 1$ to n do $C[i,j] = 0$
  for $k \leftarrow 1$ to n do
  $C[i,j] += A[i,k] \times B[k,j]$
  od od od

- Clearly,
$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = n^3.$$

- Using divide and conquer, Strassen's algorithm improves the scalar multiplication cost to be in $\Theta(n^{\log_2 7}) = \Theta(n^{2.807})$.

Partition a Matrix into Submatrices

- An order $n = 2m$ matrix X can be partitioned into four order m submatrices X_{00}, X_{01}, X_{10}, and X_{11}:

$$X = \begin{bmatrix} X_{00} & X_{01} \\ X_{10} & X_{11} \end{bmatrix}.$$

Partition a Matrix into Submatrices

- The product $C = AB$ of two order $n = 2m$ matrices A, B can be found with seven order m submatrix multiplications.
- Let

$$\begin{aligned}
M_1 &= (A_{00} + A_{11}) \times (B_{00} + B_{11}), \\
M_2 &= (A_{10} + A_{11}) \times B_{00}, \\
M_3 &= A_{00} \times (B_{01} - B_{11}), \\
M_4 &= A_{11} \times (B_{10} - B_{00}), \\
M_5 &= (A_{00} + A_{01}) \times B_{11}, \\
M_6 &= (A_{10} - A_{00}) \times (B_{00} + B_{01}), \\
M_7 &= (A_{01} - A_{11}) \times (B_{10} + B_{11}).
\end{aligned}$$

Partition a Matrix into Submatrices

- It can be verified that:

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}.$$

Strassen's Matrix Multiplication Algorithm

- Strassen's algorithm simply employs the above method recursively whenever a matrix multiplication is needed.

Number of Scalar Multiplications of Strassen's Alg.

- The recurrence for $M(n)$, the number of scalar multiplications to multiply two order n matrices, is

$$M(n) = 7M\left(\frac{n}{2}\right), \qquad M(1) = 1.$$

- If $n = 2^k$ (i.e., $k = \log_2 n$), then

$$M(n) = M(2^k) = 7M(2^{k-1}) = \ldots = 7^k = 7^{\log_2 n} = n^{\log_2 7} = n^{2.807}.$$

Number of Scalar Additions of Strassen's Alg.

- The recurrence for $A(n)$, the number of scalar additions/subtractions to multiply two order n matrices, is

$$A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2, \qquad A(1) = 0.$$

- By the Master Theorem,

$$A(n) \in \Theta(n^{\log_2 7}).$$

- Thus the counts $M(n)$ and $A(n)$ have the same order of growth, and both are better than the brute force algorithm.

The Closest Pair Problem

- Recall that given n points $P_1 = (x_1, y_1), \cdots, P_n = (x_n, y_n)$, a closest pair can be identified by finding the minimum of $\binom{n}{2}$ square distances:

$$\min_{1 \le i < j \le n} (x_i - x_j)^2 + (y_i - y_j)^2.$$

- This brute force approach is a $\Theta(n^2)$ algorithm.
- With divide and conquer, we can formulate a $\Theta(n \log n)$ algorithm.

A Divide and Conquer Closest Pair Algorithm: Idea

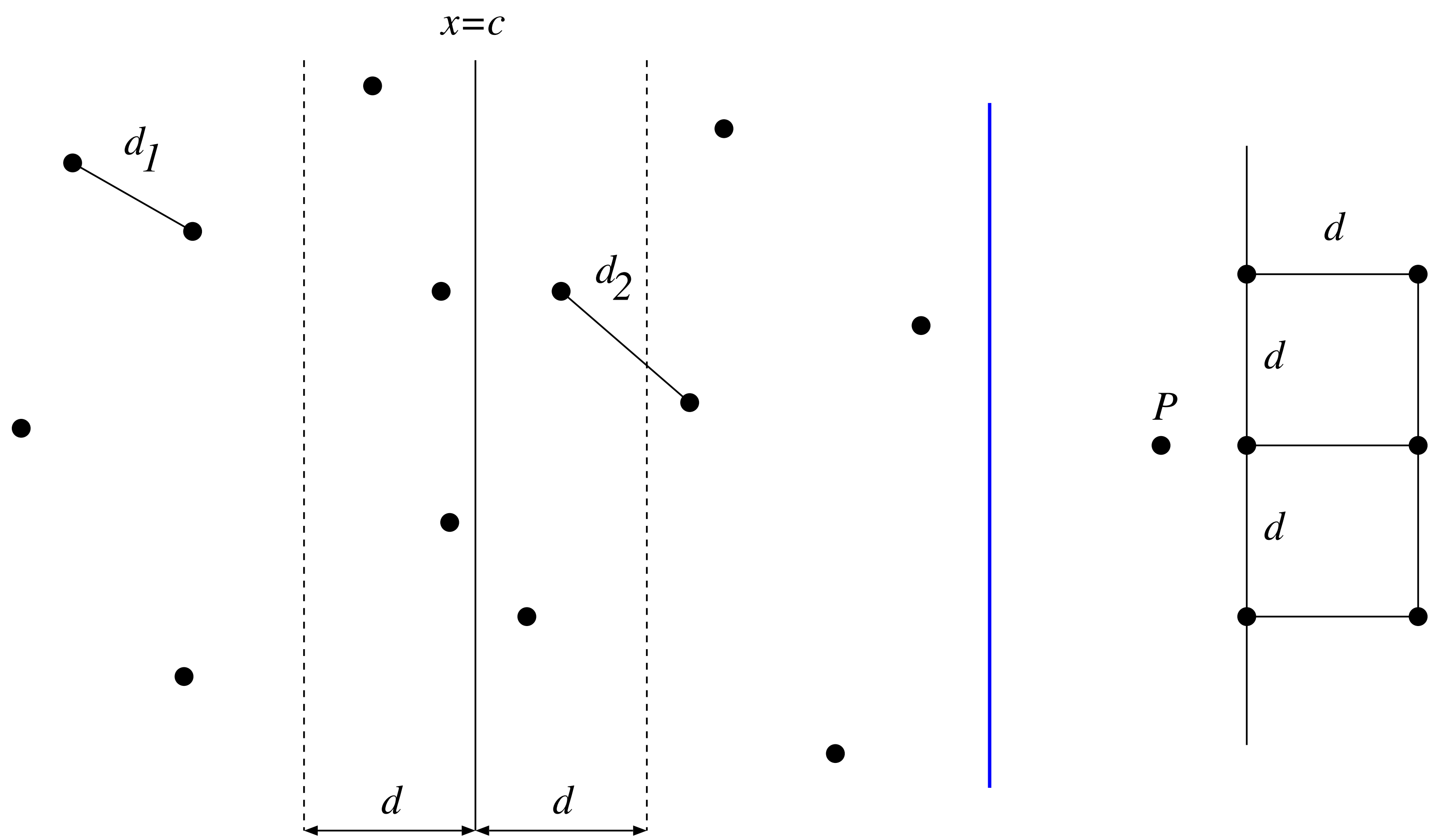
- For simplicity we may assume the number of points n is a power of 2.
- Sort the n points by their x coordinates.
- Divide the n points into two $m = \frac{n}{2}$ element sets

$$C_1 = \{P_i : 1 \leq i \leq m\} \quad \text{and} \quad C_2 = \{P_i : m+1 \leq i \leq n\}.$$

- The two groups are separated by the vertical line

$$x = \frac{x_m + x_{m+1}}{2} = V.$$

Closest Pair Explanation



A Divide and Conquer Closest Pair Algorithm: Idea

- Find the distance d_i of a closest pair in C_i for $i = 1, 2$.
- We may assume the algorithm that finds a closest pair in C_i has sorted the points in C_i by their y coordinates.
- Let $d = \min(d_1, d_2)$.
- Merge the points in C_1 and C_2 by their y coordinates so that the points are now sorted by y coordinates.
- Let
$$S = \{P_i : V - d \leq x_i \leq V + d\}.$$
- Clearly, only two points in $S \cap (C_1 \cup C_2)$ can possibly be closer than d.

A Divide and Conquer Closest Pair Algorithm: Idea

- For each $P_i \in S$, the set

$$V_i = \{P_j : y_i \leq y_j, d(P_i, P_j) \leq d, P_i \neq P_j\}$$

  contains at most six points.
- By scanning the list S, sorted by the y coordinate, from bottom to top, each time we need to check the next five points to see if there is a pair closer than d.
- Pick any of those pairs that have the smallest distance in the preceding step.

Analysis of the Divide and Conquer Closest Pair Alg.

- The main work of the algorithm is really the combination of subsolutions into a solution.
- Let this work be $M(n)$ which is in $O(n)$.
- The recurrence is thus

$$C(n) = 2C(n/2) + M(n).$$

Analysis of the Divide and Conquer Closest Pair Alg.

- By the Master Theorem we have

$$C(n) = O(n \log n).$$

- Since the sorting by x coordinates takes $\Theta(n \log n)$, so the time efficiency of the divide and conquer algorithm is $\Theta(n \log n)$.

Take Away Message on Divide-and-Conquer

- The divide-and-conquer strategy is very general and solves a problem by dividing a problem's instance into several smaller, non-overlapping instances, solving each of them recursively, and then combining their solutions to get a solution to the original instance of the problem.

- The runtime $T(n)$ of many divide-and-conquer algorithms satisfies the recurrence $T(n) = aT(n/b) + f(n)$. The Master Theorem establishes the order of growth of its solutions.