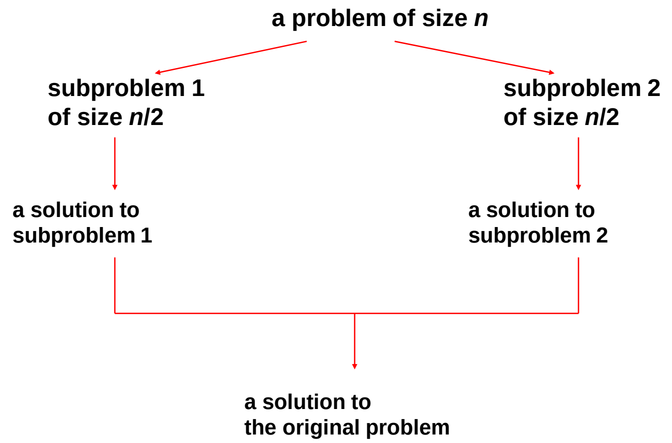
Divide-and-Conquer

- The most-well known algorithm design strategy:
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively (can implement iteratively)
- Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique



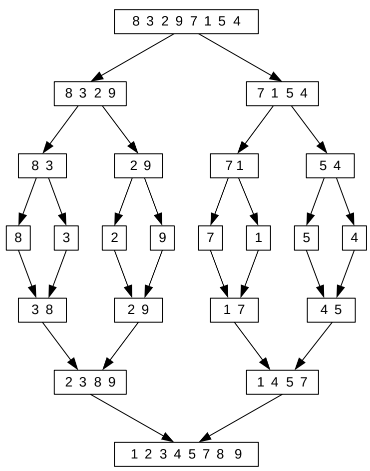
Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm
- Polynomial multiplication
- Fast Fourier Transform
- Closest-pair algorithms
- Convex-hull algorithms

Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
  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 $C[0..\lceil n/2 \rceil - 1]$
  $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
  $Mergesort(C[0..\lceil n/2 \rceil - 1])$
  $Merge(B, C, A)$

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \le C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis of Mergesort

- All cases have same efficiency: Θ(n log n)
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  - ⌈log2 n!⌉ ≈ n log2 n - 1.44n
- Space requirement: Θ(n) (not in-place)
- Can be implemented without recursion (bottom-up)

QuickSort

```
quickSort(int a[], int left, int right)   {
        // note that this works on subarray
        // defined by left and right

        if ( l < r)
                s = Partition(a, left, right);
                quickSort(a, left, s-1);
                quickSort(a, s+1, right);

  }
```


Analysis of Quicksort

- Best case: split in the middle — Θ(n log n)
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — Θ(n log n)
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion
  - These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files (n ≥ 10000)

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Examples: $T(n) = 2T(n/2) + C \Rightarrow T(n) \in ?$
What if a = 1, 4; what term dominates?

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in ?$
What if a = 4, 8; what term dominates?

$T(n) = 2T(n/2) + n^2 \Rightarrow T(n) \in ?$
What if a = 4, 8; what term dominates?

General Divide-and-Conquer Recurrence

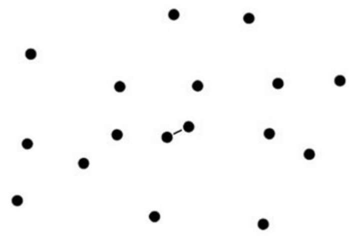
$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$ or $\log_b (a) < d$, $T(n) \in \Theta(n^d)$
If $a = b^d$ or $\log_b (a) = d$, $T(n) \in \Theta(n^d \log n)$
If $a > b^d$ or $\log_b (a) > d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Closest Pair Problem

- Naïve approach – compute distance between all pairs $\Theta(n^2)$ -- Can we do better?



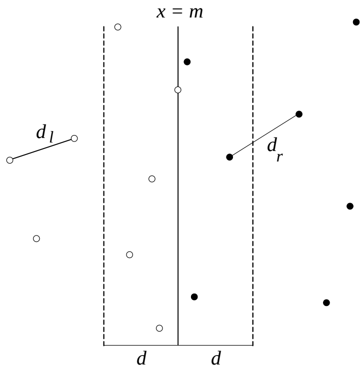
Closest-Pair Problem by Divide-and-Conquer

Step 1a: Sort the points by both x and y coordinates. Need to keep separate arrays to access the points in sorted order.

- Some algorithms only sort by x-coordinate initially, then sort by y-coordinate in the combine step

Closest-Pair Problem by Divide-and-Conquer

- Step 2: Divide the points given into two subsets P_{left} and P_{right} by a vertical line x = m so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Closest Pair by Divide-and-Conquer (cont.)

- Step 3: Find recursively the closest pairs for the left and right subsets.
- Step 4: Set d = min { d_l , d_r }

  We can limit our attention to the points in the symmetric vertical strip S of width 2d as potentially closest pair.
  (The points are stored and processed in increasing order of their y coordinates.)
- Step 5: Scan the points in the vertical strip S from the lowest up to highest. For every point p(x, y) in the strip, inspect points in the strip that may be closer to p than d. There can be no more than 5 such points following p on the strip list!

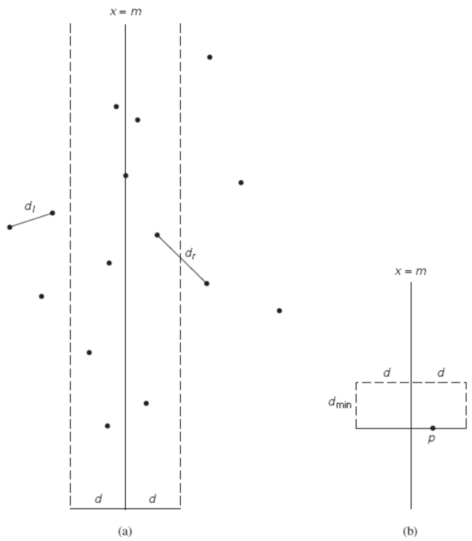


FIGURE 5.7 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem.
(b) Rectangle that may contain points closer than d_{min} to point p.

Divide and Conquer Closest-Pair Algorithm

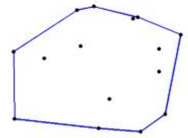
P: Array of points in non-decreasing order of x-coordinate
Q: Array of same points in non-decreasing order of y-coordinate
If $n \leq 3$
    return minimum distance using brute force
else
    copy first half of points into P_{left} , second half into P_{right}
    copy same points from Q into Q_{left} , second half into Q_{right}
            // points ordered both by x and y coordinates
    d_{left} = closestPair (P_{left} , Q_{left})
    d_{right} = closestPair (P_{right} , Q_{right})
    d_{min} = min {d_{left} , d_{right} }
    m = middle x-coordinate = P[n/2].x
    copy all the points within $2d_{min}$ band around m into a temp array sorted by y coord
    loop over the array, for each point - p
        loop over points q that are within d_{min} vertically ($|p.y – q.y| < d_{min}$)
            check if dist(p,q) smaller than d_{min},
                if yes replace d_{min} with dist(p,q)
    return d_{min}

Efficiency of the Closest-Pair Algorithm

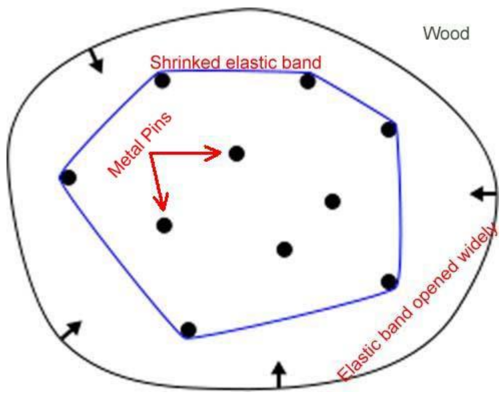
- Running time of the algorithm is described by
- $T(n) = 2T(n/2) + M(n)$, where $M(n) \in O(n)$ just like Merge sort
- By the Master Theorem (with a = 2, b = 2, d = 1)
  $T(n) \in O(n \log n)$

Convex Hull Problem: given a set of points, find the smallest convex set containing the points, Convex Hull

- A convex combination of two distinct points is any point on the line segment between them.
- Convex set: A set of points in the plane is called convex if, for any two points p and q in the set, the entire line segment from p to q including the endpoints belongs to the set.
- Convex hull of a set S is the smallest convex set that includes S
- To “solve” the convex hull problem we will find the extreme points of the convex set – that is the corners of the convex hull.



Convex Hull – Physical determination

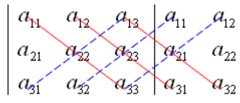


Some necessary computational facts

Two key steps: Determine

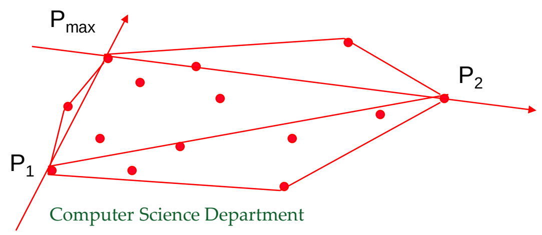
- Where a point, p_3, is to the left or right of a line (with direction), e.g. line from p_1 to p_2 .
- The distance of a point p from a line.

- Great news – both of these are solved by one calculation
  - Det of $\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$
  - Det > 0, then p_3 is to the left of $\overrightarrow{p_1p_2}$; < 0, to the right ; = 0, on the line
  - Finally normalizing by the length of the segment $\overline{p_1p_2}$ gives the distance



Quickhull Algorithm

- Convex hull: smallest convex set that includes given points
- Assume points are sorted by x-coordinate values
- Identify extreme points P_1 and P_2 (leftmost and rightmost)
- Compute upper hull recursively:
  - find point P_{max} that is farthest away from line P_1P_2
  - compute the upper hull of the points to the left of line P_1P_{max}
  - compute the upper hull of the points to the left of line $P_{max}P_2$
- Compute lower hull in a similar manner



Quickhull Algorithm

Input = a set S of n points

Assume that there are at least 2 points in the input set S of points

QuickHull (S)

// Find convex hull from the set S of n points

Convex Hull := {}

Find left and right most points, say A & B, and add A & B to convex hull

Segment AB divides the remaining (n-2) points into 2 groups S_1 and S_2

where S_1 are points in S that are on the left side of the oriented line from A to B,

and S_2 are points in S that are on the left side of the oriented line from B to A

FindHull (S_1, A, B)

FindHull (S_2, B, A)

Quickhull Algorithm

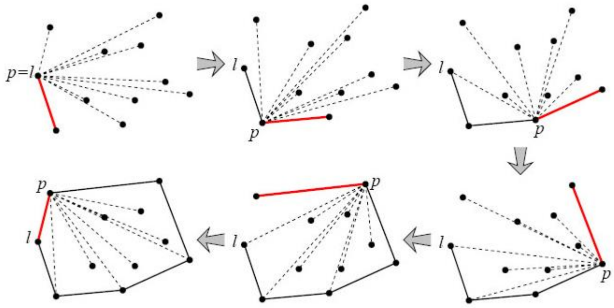
FindHull (S_k, P, Q)
// Find points on convex hull from the set S_k of points
// that are on the left side of the oriented line from P to Q
If S_k has no point then return
From the given set of points in S_k, find farthest point, say C, from segment PQ
Add point C to convex hull at the location between P and Q
Three points P, Q, and C partition the remaining points of S_k into 3 subsets:
S_0 are points inside triangle PCQ,
S_1 are points on the left side of the oriented line from P to C, and
S_2 are points on the left side of the oriented line from C to Q.
FindHull(S_1, P, C)
FindHull(S_2, C, Q)
Output = convex hull

Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- If points are not initially sorted by x-coordinate value, this can be accomplished in O(n log n) time
- Several O(n log n) algorithms for convex hull are known

Convex Hull Problem: Jarvis March (Wrapping algorithm)

Algorithm finds the points on the convex hull in the order in which they appear. It is quick if there are only a few points on the convex hull, but slow if there are many. Let x_0 be the leftmost point. Let x_1 be the first point counterclockwise when viewed from x_0. etc. (O(nh)) h: #pts in CH)



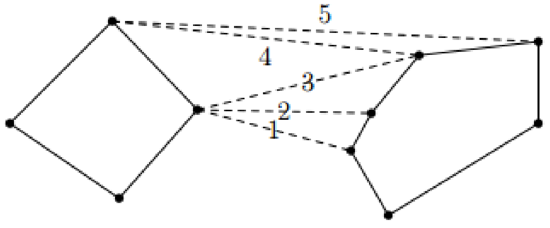
The execution of Jarvis's March.

Convex Hull Problem: (Pure) Divide and Conquer

Divide and conquer

1. Divide the n points into two halves.
2. Find convex hull of each subset.
3. Combine the two hulls into overall convex hull.

Combine! (march up/down until upper/lower tangent)



Convex Hull Problem: Graham Scan

The idea is to identify one vertex of the convex hull and sort the other points as viewed from that vertex. Then the points are scanned in order. Let x_0 be the leftmost point and number the remaining points by angle from x_0 going counterclockwise: x_1; x_2; : : : ; x_{n-1}. Let $x_n = x_0$, the chosen point.