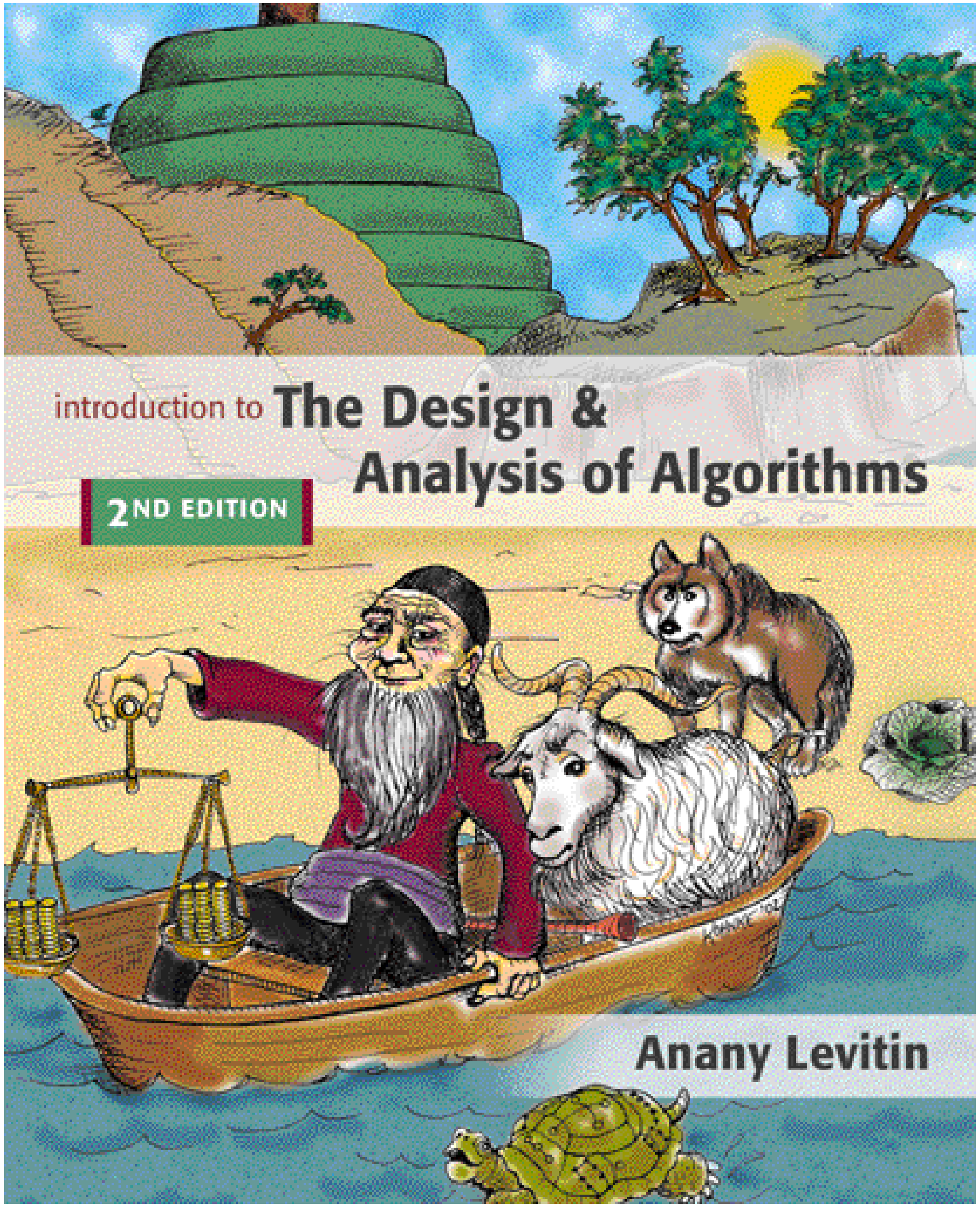
Chapter 4

Divide-and-Conquer



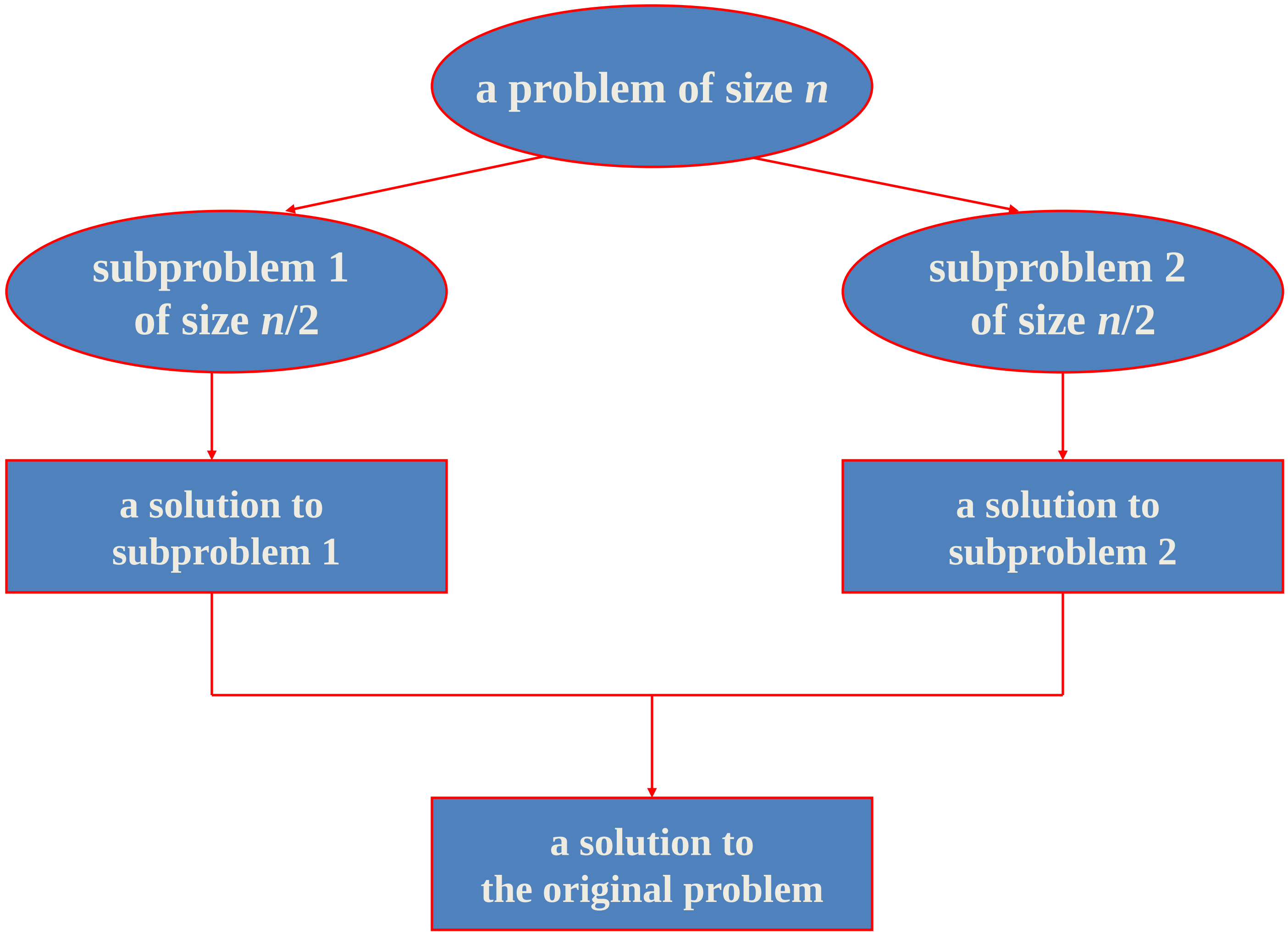
Copyright © 2007 Pearson Addison-Wesley. All rights reserved.

Divide-and-Conquer

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

- Question: always better than brute force?

Divide-and-Conquer Technique



What if using parallel computers?

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Matrix multiplication: Strassen's algorithm
- Closest-pair and convex-hull algorithms

Cases: division by 2 and general

- Typical case: division by 2

- General case:
  - Instance of length *n* is divided into *b* instances of length *b/n*, from which *a* must be solved (constants *a*>=1 and *b*>1)
  - Assuming *n* being a power of *b* (to simplify the analysis) we have

$$T(n) = a\,T(n/b) + f(n)$$

  - Where *f(n)* is the time taken to divide the problem in smaller instances and/or to combine the solutions.

General Divide-and-Conquer Recurrence

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

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log_b n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ

Examples:

$T(n) = 4T(n/4) + n \Rightarrow T(n) \in ?$

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in ?$

$\mathrm{T(n) = 2T(n/2) + 1} \Rightarrow \mathrm{T(n)} \in ?$

$\mathrm{T(n) = T(n/2) + n} \Rightarrow \mathrm{T(n)} \in ?$

See Appendix B for the proof of the theorem

General Divide-and-Conquer Recurrence

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

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log_b n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ

Examples:

$T(n) = 4T(n/4) + n \Rightarrow T(n) \in ?$ (a=4;b=4;d=1)

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in ?$ (a=2;b=2;d=1)

$\mathrm{T(n) = 2T(n/2) + 1} \Rightarrow \mathrm{T(n)} \in ?$ (a=2;b=2;d=0)

$\mathrm{T(n) = T(n/2) + n} \Rightarrow \mathrm{T(n)} \in ?$ (a=1;b=2;d=1)

General Divide-and-Conquer Recurrence

<u>Master Theorem</u>: $T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log_b n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ

Examples:

$T(n) = 4T(n/4) + n \Rightarrow T(n) \in \Theta(n^d \log_b n) = \mathbf{\Theta(n \log_4 n)}$ (a=4;b=4;d=1)

$T(n) = 2T(n/2) + n \Rightarrow T(n) \in \Theta(n^d \log_b n) = \mathbf{\Theta(n \log_2 n)}$ (a=2;b=2;d=1)

$T(n) = 2T(n/2) + 1 \Rightarrow T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \mathbf{\Theta(n)}$ (a=2;b=2;d=0)

$T(n) = T(n/2) + n \Rightarrow T(n) \in \Theta(n^d) = \mathbf{\Theta(n)}$ (a=1;b=2;d=1)

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 (total n):
    - 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.

Pseudocode of Mergesort

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

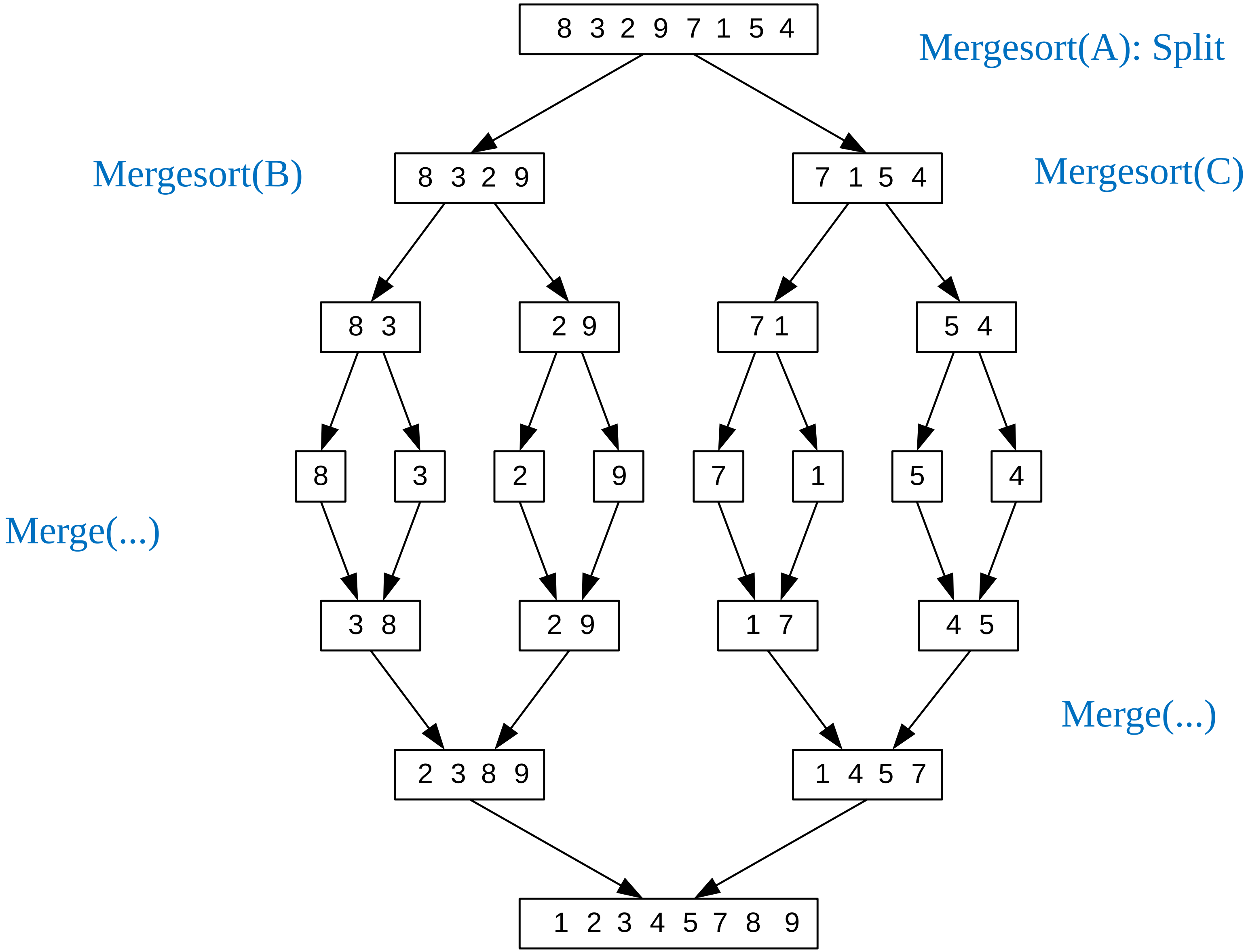
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil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

Pseudocode of Mergesort

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 ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← 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]
```

Mergesort Example



Analysis of Mergesort

- Assuming for simplicity that n is a power of 2, the recurrence relation of the number of key comparisons C(n) is

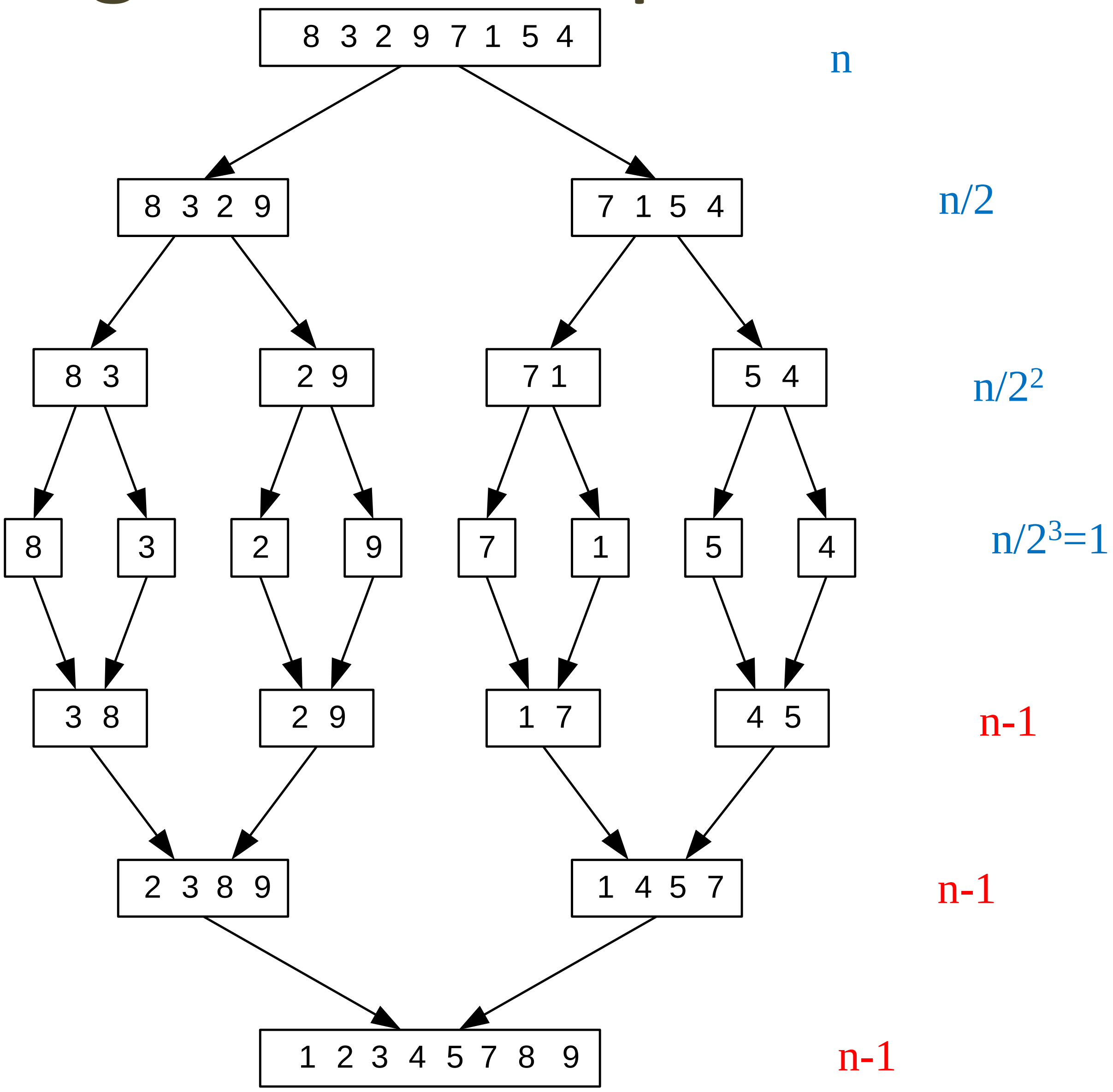
$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n>1;\ C(1) = 0$$

- $\mathbf{C_{merge}(n)}$ **:** no pior caso, cada chave vem de uma partição a cada vez ➔ n-1 comparações

$$C_{worst}(n) = 2C_{worst}(n/2) + n-1$$

Resolvendo a recorrência, ou aplicando o Master Theorem, chegamos a **O(n log n)**

Mergesort Example



0 comp. nas divisões

(n-1)*log n =
n log n – log n

Analysis of Mergesort

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n =$$

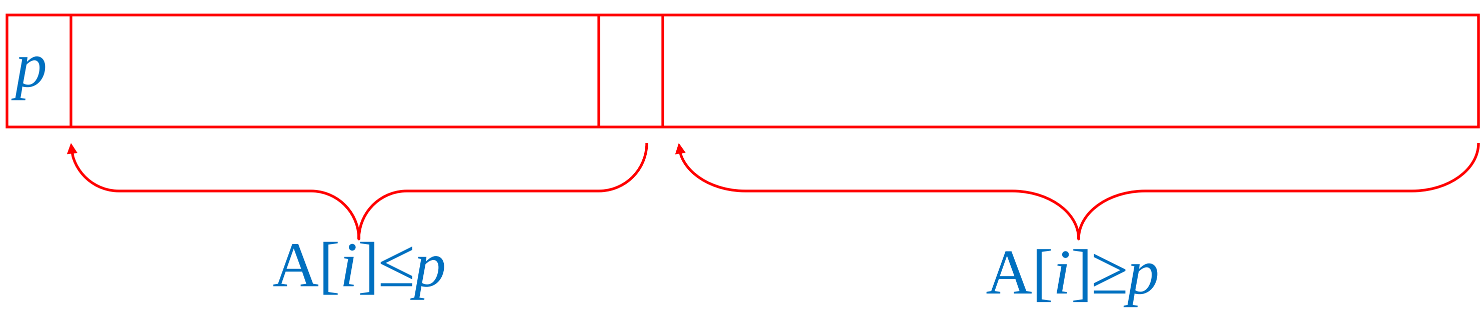
- Space requirement: $\Theta(n)$ (<u>not</u> in-place)

- Can be implemented without recursion (bottom-up)

- Stable ???

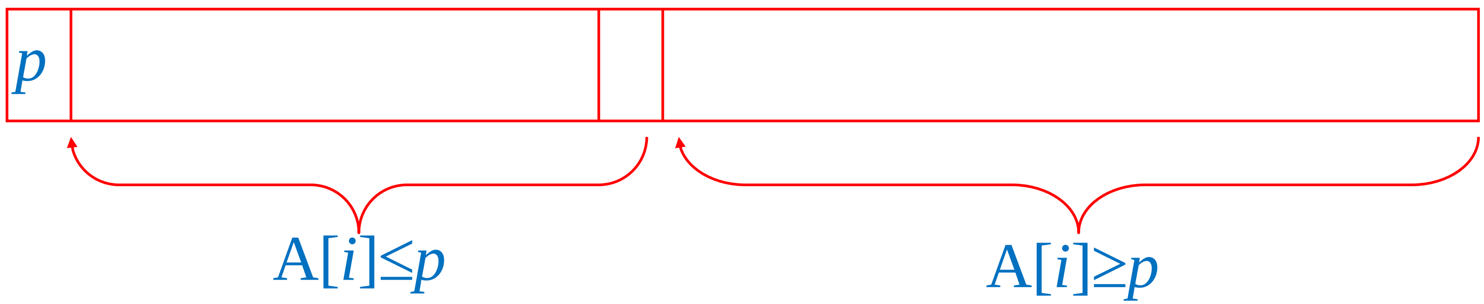
Quicksort

- Select a *pivot* (partitioning element) – here, 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* positions are larger than or equal to the pivot (see next slide for an algorithm)



- Exchange the pivot with the last element in the first (i.e., ≤) sub array — the pivot is now in its final position
- Sort the two sub arrays recursively

Quicksort Algorithm



Quicksort(A[l..r])

//Input: a sub array A[l..r] of A[0..n-1]

//Output: sub array sorted in no decreasing order

If l < r

    s ← Partition(A[l..r]) // s is a split position

    Quicksort(A[l..s-1])

    Quicksort(A[s+1..r])

Partitioning Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
//   indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
//   this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\quad j \leftarrow r + 1$
repeat
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Quicksort Examples

5 3 1 9 8 2 4 7

3 4 5 6 7

Stable???

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
  - n+1 if indices cross; n if they coincide
  - $C_{best}(n) = 2\, C_{best}(n/2) + n$ p/ $n>1$, $C_{best}(1)=0$ *(Master Theorem)*

- Worst case: sorted array and pivot A[0] — $\Theta(n^2)$
  - $C_{worst}(n) = (n+1) + n + \ldots + 3 = ((n+1)(n+2)/2) - 3$

- Average case: random arrays — $\Theta(n \log n)$
  - $C_{avg}(n) = 1/n \sum_{s=0}^{s=1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$ p/ $n>1$, $C_{avg}(0)=0$, $C_{avg}(1)=0$
  - $C_{avg}(n) \approx 2n \ln n \approx 1.38n \log n$ (38% more than the best case)

Analysis of Quicksort

- Improvements :
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small sub files
  - elimination of recursion
  - In combination, they improve by 20-25%

- Considered the method of choice for internal sorting of large files ($n \geq 10000$)

Binary Search

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

```
l ← 0;   r ← n-1
while l ≤ r do
    m ← ⌊(l+r)/2⌋
    if  K = A[m]  return m
    else if K < A[m]  r ← m-1
    else l ← m+1
return -1
```

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor),\ C_w(1) = 1$

    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

- Bad (degenerate) example of divide-and-conquer (Decrease-by-half algorithm)

Binary Tree Algorithms

Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

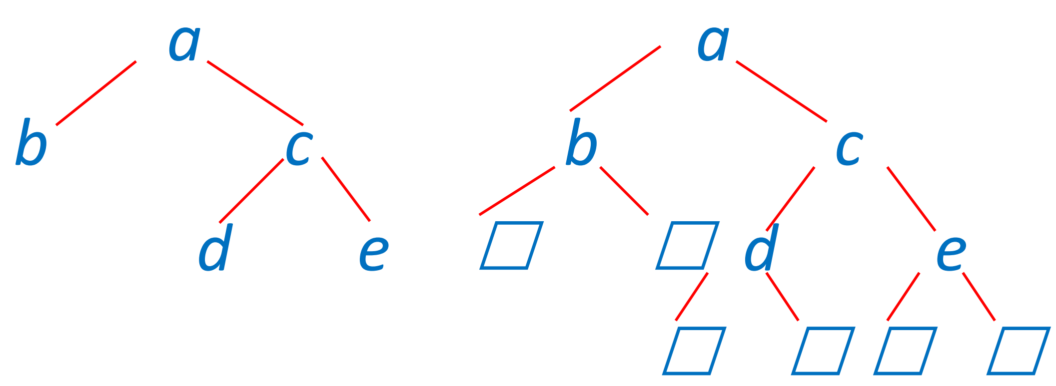
Algorithm *Inorder*(T)

if $T \neq \varnothing$

  Inorder(T_{left})

  print(root of T)

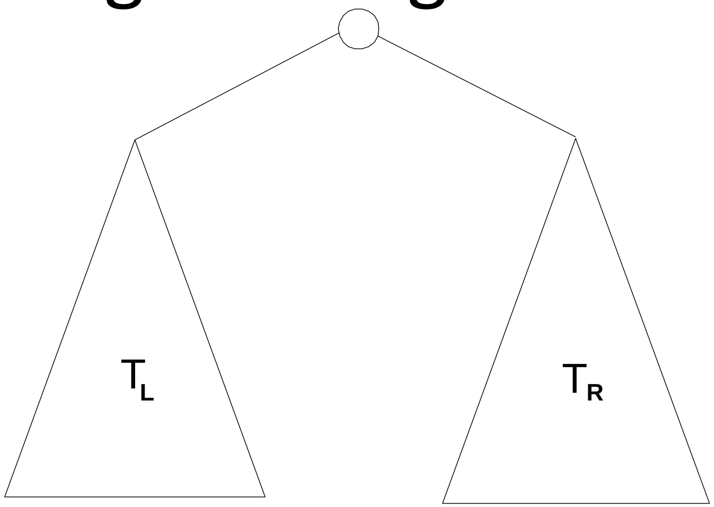
  Inorder(T_{right})



Efficiency: **Θ(*n*)**

Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree



$$h(T) = \max\{h(T_L), h(T_R)\} + 1 \text{ if } T \neq \varnothing \text{ and } h(\varnothing) = -1$$

Efficiency: $\Theta(n)$

Strassen's Matrix Multiplication

Strassen [1969] observed that the product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

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

$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$

$M_2 = (A_{10} + A_{11}) * B_{00}$

$M_3 = A_{00} * (B_{01} - B_{11})$

$M_4 = A_{11} * (B_{10} - B_{00})$

$M_5 = (A_{00} + A_{01}) * B_{11}$

$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$

$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$

Requires 7 products and 18 add./subtr. while brute force requires 8 prod. and 4 add.

Analysis of Strassen's Algorithm

Let A and B n-by-n matrices where n is a power of 2 (If n is not a power of 2, matrices can be padded with zeros)

Number of multiplications:

$$M(n) = 7M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 7^{\log n} = n^{\log 7} \approx \boldsymbol{n^{2.807}}$

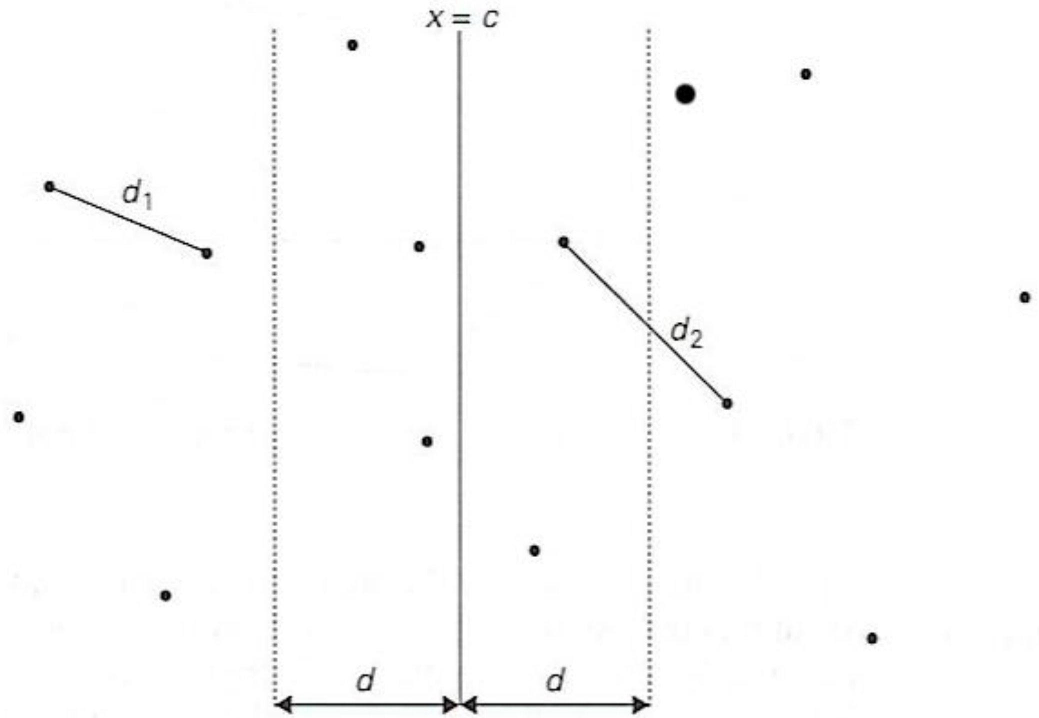
vs. $\boldsymbol{n^3}$ of brute-force alg.

Algorithms with better asymptotic efficiency ($\boldsymbol{n^{2.376}}$) are known but they are even more complex.

- Lower Bound = $\boldsymbol{n^2}$

Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets S_1 and S_2 by a vertical line $x = c$ 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.



- **We can assume that the points are ordered on x coordinates (may use Mergesort, O(nlogn)).**
- **Can use as c the median of x coordinates**

Closest Pair by Divide-and-Conquer

Step 2 Find <u>recursively</u> the closest pairs for the left (d_1) and right (d_2) subsets.

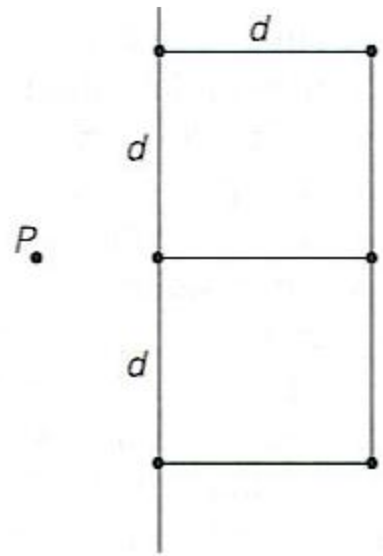
Step 3 Set $d = \min\{d_1, d_2\}$

We can limit our attention to the points in the symmetric vertical strip of **width 2d** as possible closest pair. Let C_1 and C_2 be the subsets of points in the left subset S_1 and of the right subset S_2, respectively, that lie in this vertical strip. The points in C_1 and C_2 are stored in increasing order of their y coordinates, which is maintained by merging during the execution of the next step.

Step 4 For every point $P(x,y)$ in C_1, we inspect points in C_2 that may be closer to P than d.

Closest Pair by Divide-and-Conquer: Worst Case

The worst case scenario is depicted below:



Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

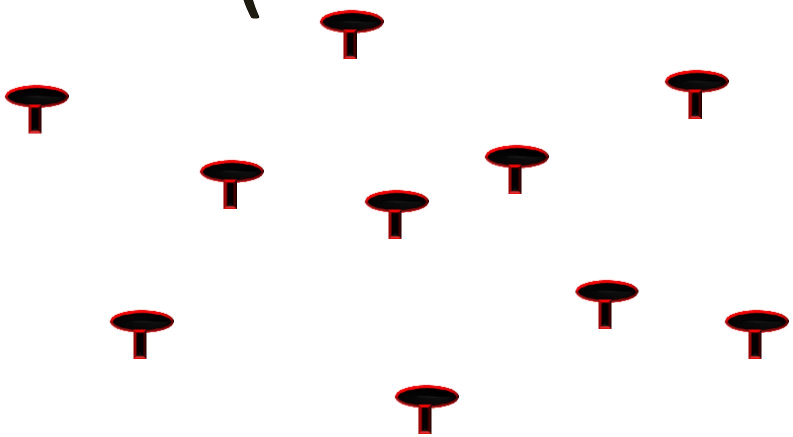
(the time M(n) for merging solutions is O(n))

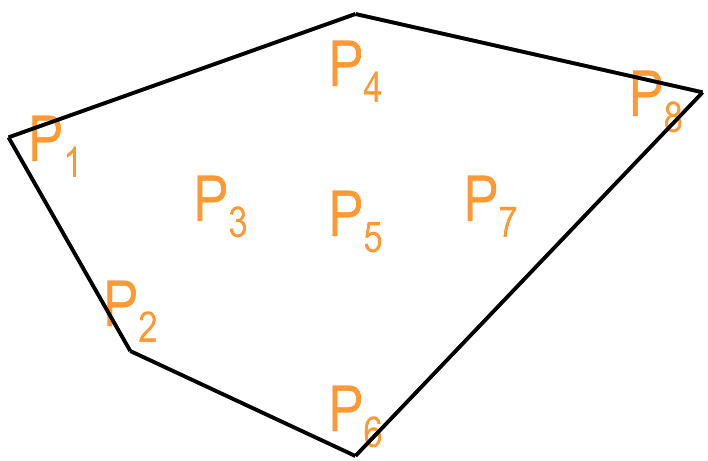
$$T(n) = 2T(n/2) + M(n), \quad \text{where } M(n) \in O(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$\mathbf{T(n) \in O(n \log n)}$$

Convex hull
(envoltória convexa)





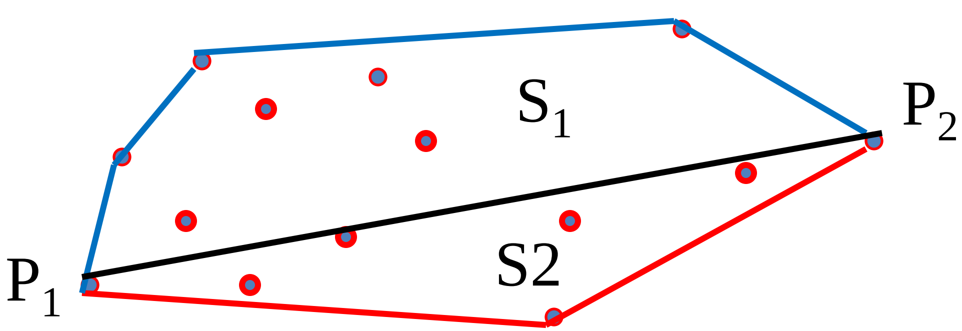
Smallest convex polygon that contains n given points in the plane (for any 2 points, there is a line inside the polygon)

Brute Force: $O(n^3)$

Quickhull Algorithm

***Convex hull*: smallest convex polygon that includes given points S**

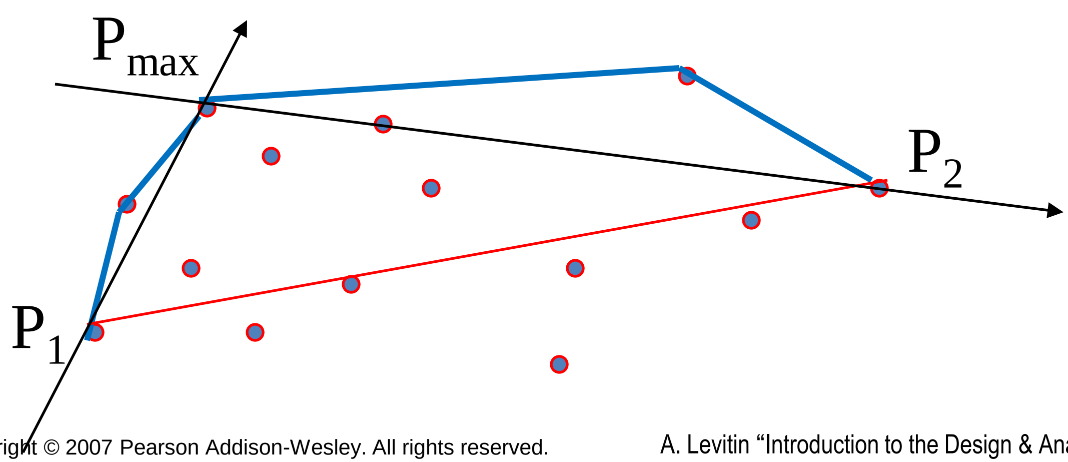
- Assume points are sorted by *x*-coordinate values
- Identify *extreme points* P_1 and P_2 , the leftmost (lowest x) and rightmost (largest x)
- $\overrightarrow{P_1 P_2}$ divides S in S_1 (points to left) and S_2 (points to the right)
- **Upper Hull**: line with vertices at P_1 , some of points in S_1 and P_2. If S_1 is empty, it is the line $P_1 P_2$
- **Lower Hull:** line with vertices at P_1 , some of points in S_2 and P_2. If S_2 is empty, it is the line $P_1 P_2$



Quickhull Algorithm

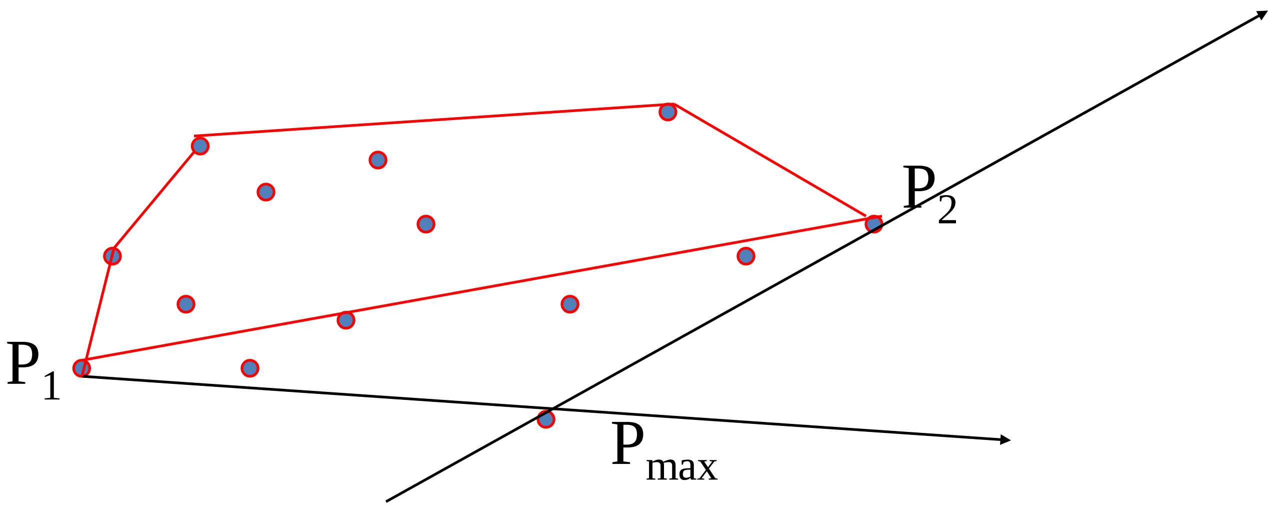
How to compute upper hull recursively :

- 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 $P_{max}P_2$



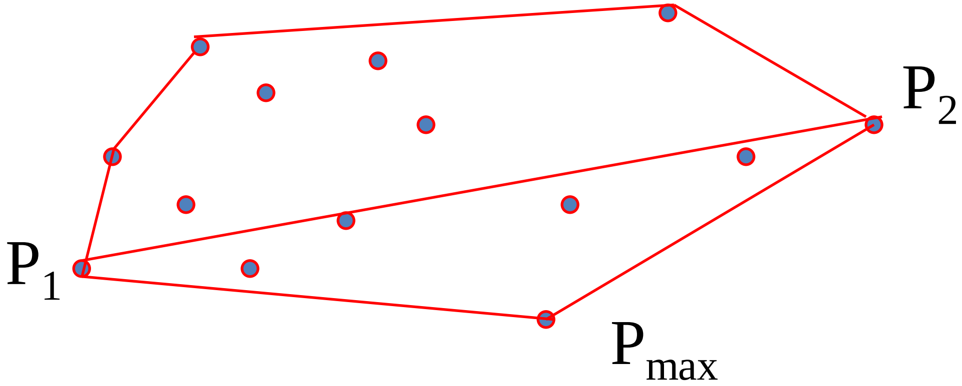
Quickhull Algorithm

- Compute *lower hull* recursively (similar way):



Quickhull Algorithm

- Compute *lower hull* recursively (similar way):



Efficiency of Quickhull Algorithm (as Quicksort)

- If points are not initially sorted by *x*-coordinate value, this can be accomplished in O(*n* log *n*) time.

$$T(n) = T_{uh}(x) + T_{lh}(y) + T(finding\_point\_farthest\_away);\ x+y = n$$

- T(finding_point_farthest_away) = O(n)
- If x or y always = 1: Worst Case (as Quicksort)

$$T(n) = T_{uh}(n-1) + n \Rightarrow \Theta(n^2)$$

- If divides always into 2 halves: Best Case

$$T(n) = 2T_{u\&lh}(n/2) + n \Rightarrow \Theta(n \log n)$$

- Average case: **Θ(*n*)** (under reasonable assumptions about distribution of points given)

- Several O(*n* log *n*) algorithms for convex hull are known

Efficiency of Quickhull Algorithm

Then:

Total time = Sorting time + Recursion time

$n \log n + n \log n$

$= O(n \log n)$