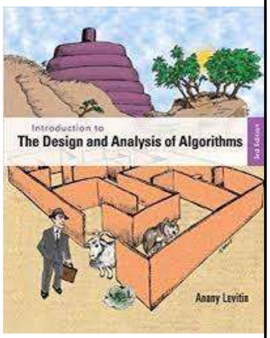
5-Divide-and-Conquer

A. Levitin "Introduction to the Design & Analysis of Algorithms," 3rd ed., Ch. 1 ©2012 Pearson Education, Inc. Upper Saddle River, NJ. All Rights Reserved



1

Divide-and-Conquer

The most-well known algorithm design strategy:

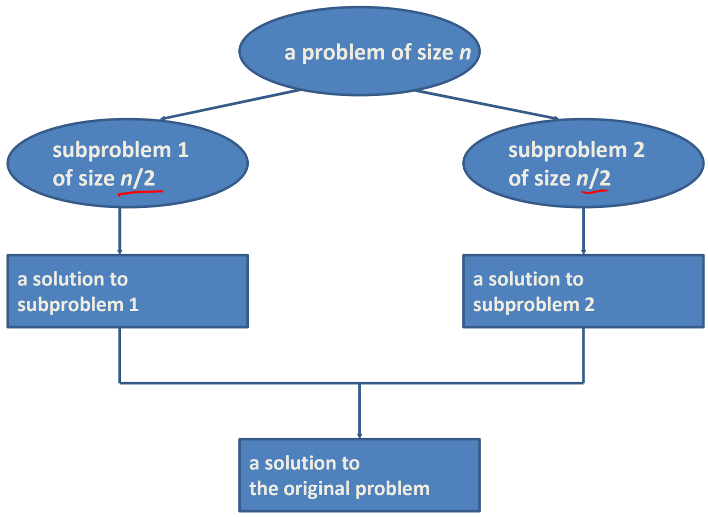
1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

2

Divide-and-Conquer Technique (cont.)



3

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms
- *Binary search: decrease-by-half (or degenerate divide&conq.)*

4

General Divide-and-Conquer Recurrence

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

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

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

Examples: $T(n) = 4T\left(\frac{n}{2}\right) + n \rightarrow T(n) \in ?$
$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \rightarrow T(n) \in ?$
$T(n) = 4T\left(\frac{n}{2}\right) + n^3 \rightarrow T(n) \in ?$

5

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

1) $T(n) = 4T(n/2) + n$
$a = 4$ $b = 2$ $d = 1$ $b^d = 2$ $\rightarrow \log_2 2 = 2$
$a > b^d$ $\Rightarrow T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$

2) $T(n) = 4T(n/2) + n^2$
$a = 4$ $b = 2$ $d = 2$ $b^d = 4$
$a = b^d$ $\Rightarrow T(n) \in \Theta(n^2 \log n)$

6

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

3) $T(n) = 4T(n/2) + n^3$
$a=4$ $b=2$ $d=3$
$a < b^d \Rightarrow T(n) \in \Theta(n^3)$

4) $T(n) = 3T\left(\frac{2n}{3}\right) + n$ $n/3/2$
$a=3$ $b=\frac{3}{2}$ $d=1$
$a > b^d$ $T(n) \in \Theta\left(n^{\log_{3/2} 3}\right)$

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.

Pseudocode of Mergesort

$T(n) = 2T(n/2) + f(n)$

$f(n) \in \Theta(n)$

in all cases

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 $C[0..\lceil n/2 \rceil - 1]$
$\rightarrow$ *Mergesort*$(B[0..\lfloor n/2 \rfloor - 1])$
$\rightarrow$ *Mergesort*$(C[0..\lceil n/2 \rceil - 1])$
Merge(B, C, A) //see below $\rightarrow \Theta(n)$

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
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
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 $A[k..p+q-1]$

Merge

B → 1 3 5 7 9 11

C → 2 4 6 8 10 12

1 2 3

for n → n-1 comparisons

worst case $\Theta(n)$

best case

1 2 3 4 5 6

7 8 9 10 11 12

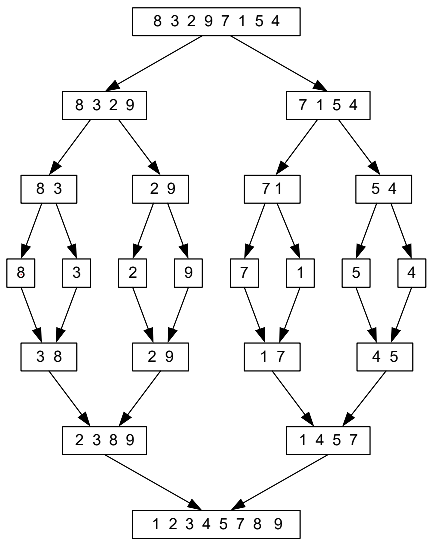
$n/2$ comparisons

$\Theta(n)$

Merge

11

Mergesort Example



12

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$
- 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: $\Theta(n)$ (not in-place)
- Can be implemented without recursion (bottom-up)

$T(n) = 2T(n/2) + f(n)$ $\quad$ $f(n) \in \Theta(n)$

Master theorem

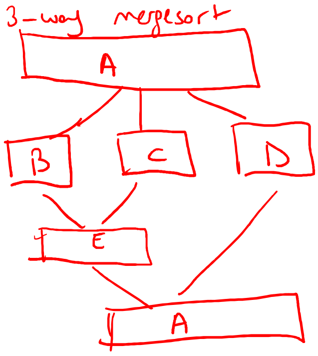
$a=2 \quad b=2 \quad d=1$

$a = b^d \Rightarrow \Theta(n^d \log n)$

$\Rightarrow \boxed{T(n) \in \Theta(n \log n)}$

in all cases.

3-way mergesort



pseudocode:
mergesort3(A[0...n-1])
if n>1
copy $A[0 \dots \lfloor n/3 \rfloor - 1]$ to B
copy $A[\lfloor n/3 \rfloor \dots 2\lfloor n/3 \rfloor - 1]$ to C
copy $A[2\lfloor n/3 \rfloor \dots n-1]$ to D
- mergesort3(B)
- mergesort3(C)
- mergesort3(D)

worst case.
- Merge(B, C, E) $\rightarrow \frac{n}{3} + \frac{n}{3} - 1 = \frac{2n}{3} - 1$ ($n/3$, $n/3$)
- merge(E, D, A) $\frac{2n}{3} + \frac{n}{3} - 1 = n - 1$ ($\frac{2n}{3}$, $\frac{n}{3}$)

Recurrence Relation

$T(n) = 3T(n/3) + f(n)$

$T(n) \leq 3T(n/3) + \underline{\frac{5n}{3} - 2}$

worst case.

$\xrightarrow{\text{M.T.}}$ $a = 3$ $b = 3$ $d = 1$

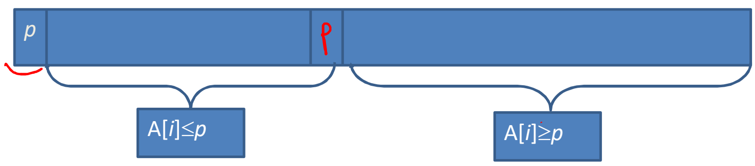
$a = b^d \Rightarrow T(n) \in \Theta(n \log n)$

9.11.2021 BLM19307E Algorithm Analysis and Design 15

15

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)

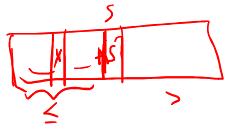


- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

9.11.2021 BLM19307E Algorithm Analysis and Design 16

16

Pseudocode of Quicksort

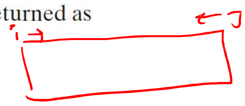


ALGORITHM *Quicksort*$(A[l..r])$

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

17

Hoare's Partitioning Algorithm



ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
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

18

Quicksort Example

5 3 1 9 8 2 4 7

5 3 1 4 8 2 9 7

5 3 1 4 2 8 9 7

5 | 3 1 4 2 | 8 9 7

2 3 1 4 | 5 | 8 9 7

1 | 2 | 3 4 5 | 7 | 8 | 9

ALGORITHM *HoarePartition*($A[l..r]$)
//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r ($l < r$)
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
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

Analysis of Quicksort

partitioning

Always $C_B(n) = 2C_B(n/2) + n$

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$ $\Rightarrow C_W(n) = C_W(n-1) + n$
- Average case: random arrays — $\Theta(n \log n)$ $\Rightarrow \Theta(n^2)$

1 2 3 4 5 6 7 8 9

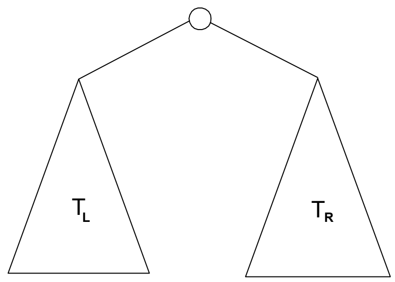
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

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

Binary Tree Algorithms

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



Binary Tree Algorithms

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

- In the **preorder** traversal, the root is visited before the left and right subtrees are visited (in that order).
- In the **inorder** traversal, the root is visited after visiting its left subtree but before visiting the right subtree.
- In the **postorder** traversal, the root is visited after visiting the left and right subtrees (in that order).

- Efficiency: $\Theta(n)$

Binary Tree Algorithms (cont.)

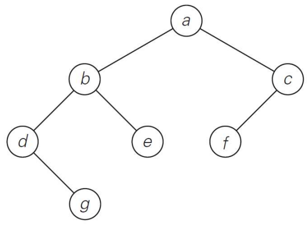
Algorithm *Inorder*(T)

if $T \neq \varnothing$

$\quad$ *Inorder*(T_{left})

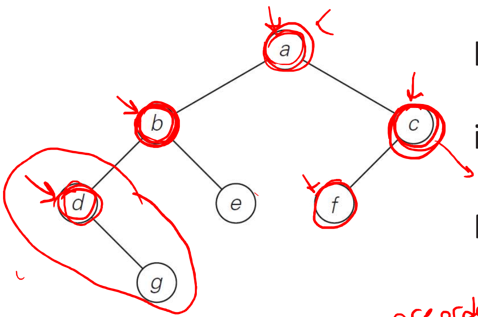
$\quad$ print(root of T)

$\quad$ *Inorder*(T_{right})



23

Binary Tree Algorithms (cont.)



N LR

preorder (Root, Left, Right)

LNR

inorder (Left, Root, Right)

Postorder (Left, Right, Root)

inorder dgbeafc

preorder a bdg e cf

postorder gd e b fc a

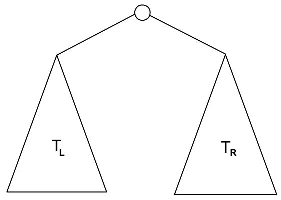
L R

L R

24

Binary Tree Algorithms (cont.)

Ex. 2: Computing the height of a binary tree



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

Efficiency: $\Theta(n)$

25

Binary Tree Algorithms (cont.)

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \emptyset$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

26

Multiplication of Large Integers

Consider the problem of multiplying two (large) n-digit integers represented by arrays of their digits such as:

A = 12345678901357986429   B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots a_n \\
b_1\ b_2 \ldots\ b_n \\
\hline
(d_{10})\ d_{11}d_{12} \ldots d_{1n} \\
(d_{20})\ d_{21}d_{22} \ldots d_{2n}\quad\ \ \\
\ldots \ldots \ldots \ldots \ldots \ldots \ldots\qquad\quad \\
(d_{n0})\ d_{n1}d_{n2} \ldots d_{nn}\qquad\qquad\ \
\end{array}
$$

Efficiency: n^2 one-digit multiplications

9.11.2021 BLM19307E Algorithm Analysis and Design 27

27

First Divide-and-Conquer Algorithm

A small example: A ∗ B where A = 2135 and B = 4014

A = (21·10² + 35), B = (40 ·10² + 14)

So, A ∗ B = (21 ·10² + 35) ∗ (40 ·10² + 14)

= 21 ∗ 40 ·10⁴ + (21 ∗ 14 + 35 ∗ 40) ·10² + 35 ∗ 14

In general, if A = A_1A_2 and B = B_1B_2 (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$$\boxed{A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2}$$

Recurrence for the number of one-digit multiplications M(n):

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: M(n) = n^2

9.11.2021 BLM19307E Algorithm Analysis and Design 28

28

Second Divide-and-Conquer Algorithm

A = $A_1 A_2$
B = $B_1 B_2$

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$
which requires only 3 multiplications at the expense of (4-1) extra add/sub.

$A * B = A_1 B_1 10^n + [(A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2] \cdot 10^{n/2} + A_2 B_2$

Recurrence for the number of multiplications M(n):

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

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$ $< \Theta(n^2)$

master theorem

Example of Large-Integer Multiplication

2135 ∗ 4014

Strassen's Matrix Multiplication

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

$A_{n \times n}$ $A_{n/2 \times n/2}$

$B_{n \times n}$ $B_{n/2 \times n/2}$

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

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

Formulas for Strassen's Algorithm

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

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

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

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

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

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

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

Analysis of Strassen's Algorithm

If n is not a power of 2, matrices can be padded with zeros.

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 6 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Number of multiplications:

$\boxed{M(n) = 7M(n/2)}$, $M(1) = 1$

Solution: $M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

$a=7 \quad b=2 \quad d=0$

master theorem.

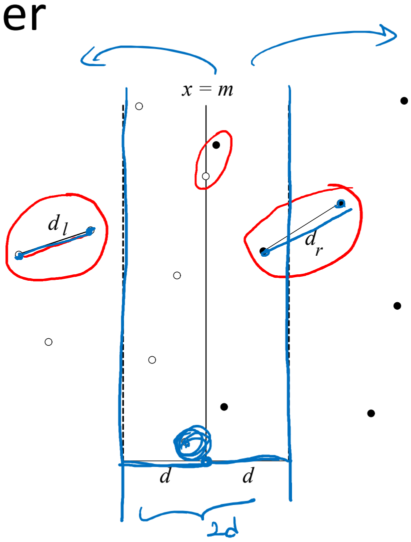
$\Theta(n^{2.807}) < \Theta(n^3)$

Algorithms with better asymptotic efficiency are known but they are even more complex.

Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets P_l and P_r 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.

$d = \min\{d_l, d_r\}$



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

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 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 possible closest pair. (The points are stored and processed in increasing order of their y coordinates.)

Step 4 Scan the points in the vertical strip S from the lowest up. For every point $p(x,y)$ in the strip, inspect points in 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!

5

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

$$T(n) = 2T(n/2) + \overset{5n}{M(n)}, \text{ where } M(n) \in O(n)$$

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

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

Quickhull Algorithm

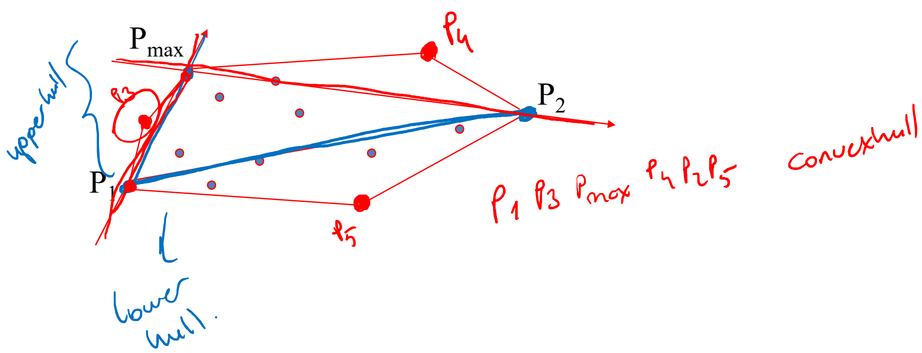
Convex hull: smallest convex set that includes given points

- 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

9.11.2021 BLM19307E Algorithm Analysis and Design 37

37

Quickhull Algorithm



9.11.2021 BLM19307E Algorithm Analysis and Design 38

38

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) → all the points are in the lower hull
  - 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

9.11.2021 BLM19307E Algorithm Analysis and Design 39

39

Example

Berna
Berila

$LCPr = \underbrace{\text{Berna}}_{m}$
$LCPL = \underbrace{\text{Zuhi}}_{m}$

Divide and Conquer Example – Longest Common Prefix

Given n strings, the problem is to find the longest common prefix of these strings. As an example, if the input is {"abdullah", "abdi", "abdal", "abdulkerim"}, output would be "abd". If the input is {"kelam", "kelime", "kemal", "kemik"}, output would be "ke".

kel kem n = 4

(a) Design a divide-and-conquer algorithm for this problem. Please provide a step by step description of your algorithm.

m = 6

(b) What is the time complexity of your algorithm? Write a recurrence relation and solve it. Provide an answer in terms of n and m, where n is the number of strings and m is the length of the largest string.

$T(n) = 2T(n/2) + m$ in the worst case worst case.
$T(1) = 0$

9.11.2021 BLM19307E Algorithm Analysis and Design 40

40

$S_1 \; S_2 \; \ldots \; S_{n/2} \;|\; S_{n/2+1} \; \ldots \; S_n$ n strings.

$\underbrace{S_1 \; S_2 \; \ldots \; S_{n/2}}_{LCP_l}$ $\underbrace{S_{n/2+1} \; \ldots \; S_n}_{LCP_r}$ 'berna'

How can combine this?

$LCP(LCP_l, LCP_r)$

if $n=1$, return whole string

else Divide n strings into two parts.

find LCP from the left part recursively.

find LCP " " right " "

find LCP of these two prefixes

and return this.

$T(n) = 2T(n/2) + m$ $n = 2^k$ $T(1) = 0$

$= 2 \cdot T(2^{k-1}) + m$

$= 2 \cdot [2T(2^{k-2}) + m] + m = 2^2 \cdot T(2^{k-2}) + 2m + m$

$= 2 \cdot [2 \cdot [2 \cdot T(2^{k-3}) + m] + m] + m$

$= 2^3 \cdot T(2^{k-3}) + 2^2 m + 2m + m$

$\vdots$

$= 2^k \, T(\underbrace{2^{k-k}}_{0}) + 2^{k-1} m + 2^{k-2} m + \ldots + 2m + m$

$= m(2^{k-1} + 2^{k-2} + \ldots + 2^1 + 1) = m(2^k - 1)$

$= m \cdot (n-1)$

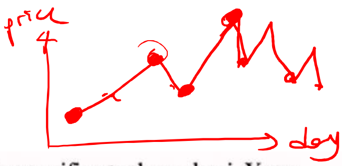
worst case. $\Rightarrow \Theta(mn)$

A[1]=50
A[2]=55

Brute force.

$\Theta(n^2)$

Example



You are given an array A of n numbers. A[i] is the price of a specific stock on day i. Your goal is to figure out the best days to buy and sell that stock in order to maximize profit. That is, you must find two days i, j such that i < j, and A[j] - A[i] is maximized. Brute-force algorithm for this problem has a complexity of $O(n^2)$, but we are seeking for a more efficient algorithm.

(a) Here is a divide and conquer algorithm for this problem:

1. If array has at most 2 elements, solve the problem trivially, return.

2. Solve the problem in first n/2 elements (recursively). Let p_1 be the maximum profit. left

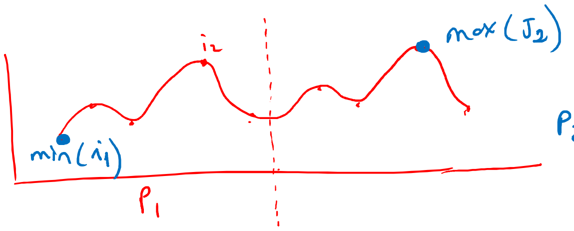
3. Solve the problem in last n/2 elements (recursively). Let p_2 be the maximum profit. right.

4. ....

....

5. Return the maximum of p_1, p_2 and p_3.

Fill the fourth step of the above algorithm.



P_3 = max-min

$i < J$

P1: max profit when I buy in the first part and sell it in the first part.

P2: max profit when I buy in the second part and sell it in the second part.

P3: max profit when I buy in the first part and sell it in the second part.

4) find min element in the (first) left part $n/2$
find max element in the right (second) part $n/2$

$i < j$

$p_3 = max - min$

recursive

4th step.

Time complexity

$$T(n) = 2T(n/2) + \underbrace{n/2}_{\text{find min}} + \underbrace{n/2}_{\text{find max}}$$

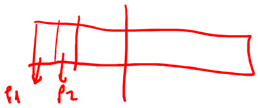
$$T(n) = 2T(n/2) + n$$

$a = 2$
$b = 2$
$d = 1$

master theorem

$T(n) \in \Theta(n \log n)$

Brute-force $\Theta(n^2)$



$T(n) = aT(n/b) + f(n)$

$f(n) \in \Theta(n^d)$

45