Chapter 4: Divide and Conquer

Divide each difficulty into as many parts as is feasible and necessary to resolve it.
(Rene Descartes)

Introduction

Divide-and-conquer is an approach to solving a problem by:

- Divide an instance into smaller instances of the problem.
- Solve the smaller instances.
- Combine the solutions of the smaller instances into a solution for the larger instance.

Illustration

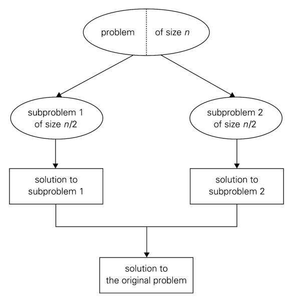


FIGURE 4.1 Divide-and-conquer technique (typical case)

General Divide-and-Conquer Recurrence

The general case is to divide an instance of size n into a instances of size n/b, taking $f(n)$ time to divide and combine.

$T(n)$	$=$	a	$T(n/b)$	$+$	$f(n)$
time for size n		number of subinstances	time for size n/b		time for divide & combine

Some examples are:

Mergesort	$T(n) = 2T(n/2) + \Theta(n)$
Binary Search	$T(n) = T(n/2) + \Theta(1)$
Strassen's	$T(n) = 7T(n/2) + \Theta(n^2)$

The Master Theorem

For the recurrence $T(n) = aT(n/b) + f(n)$, if $f(n) \in \Theta(n^d)$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Note: $T(n) \geq f(n)$. Also $T(n) \geq a^{\log_b n} = n^{\log_b a}$ because recurrence tree has at least $a^{\log_b n}$ nodes. Examples:

Mergesort	$T(n) = 2T(n/2) + \Theta(n)$	$\Theta(n \log n)$
Binary Search	$T(n) = T(n/2) + \Theta(1)$	$\Theta(\log n)$
Strassen's	$T(n) = 7T(n/2) + \Theta(n^2)$	$\Theta(n^{\log_2 7})$

Mergesort 6

Mergesort Algorithm

algorithm $Mergesort(A[0..n-1])$
// Sorts a given array by mergesort
// Input: An array A of orderable elements
// Output: Array $A[0..n-1]$ in ascending order
if $n \leq 1$ **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 $C[0..\lceil n/2 \rceil - 1]$
$Mergesort(B)$
$Mergesort(C)$
$Merge(B, C, A)$

Merge Algorithm

algorithm $Merge(B[0..p-1], C[0..q-1],$
$A[0..p+q-1])$
// Merges two sorted arrays into one array
// Input: Sorted arrays B and C
// Output: Sorted array A
$i \leftarrow 0$; $j \leftarrow 0$
for $k \leftarrow 0$ **to** $p+q-1$ **do**
  if $i < p$ **and** $(j = q$ **or** $B[i] \leq C[j])$ **then**
    $A[k] \leftarrow B[i]$; $i \leftarrow i+1$
  else
    $A[k] \leftarrow C[j]$; $j \leftarrow j+1$

Illustration

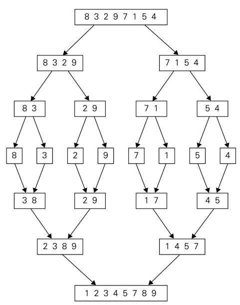


FIGURE 4.2 Example of mergesort operation

Comments on Mergesort

- Note that $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$ in *Mergesort*.
- Note that $0 \leq i \leq p$, $0 \leq j \leq q$, and $i + j = k$ in *Merge* algorithm. Comparison only happens if both $i < p$ and $j < q$.
- 2 recursive calls to *Mergesort* on instances of size $n/2$. Copy and merge is $\Theta(n)$.
- Constant factor can be improved by more efficient use of arrays.

Analysis of Mergesort

- The number of comparisons $C(n)$ is:

$$C(n) \leq 2C(n/2) + n - 1$$

- Base cases for $n =$ a power of 2: $C(1) = 0$, $C(2) = 1$, $C(4) \leq 5$, $C(8) \leq 17$.
- Induction assuming $C(n/2) \leq (n/2)\log_2(n/2)$

$$\begin{aligned} C(n) &\leq 2C(n/2) + n - 1 \\ &\leq 2(n/2)\log_2(n/2) + n - 1 \\ &\leq n\log_2(n/2) + n - 1 \\ &\leq n\log_2 n - n\log_2 2 + n - 1 \\ &\leq n\log_2 n - n + n - 1 \\ &\leq n\log_2 n \end{aligned}$$

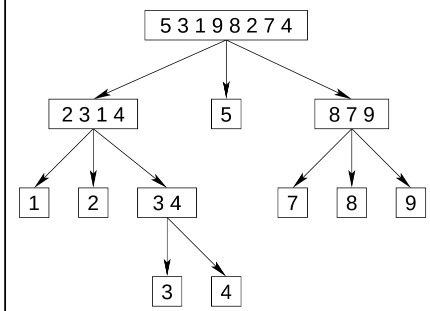
Quicksort Algorithm

algorithm $\mathit{Quicksort}(A[l..r])$
    // Sorts a subarray by quicksort
    // Input: An subarray of A
    // Output: Array $A[l..r]$ in ascending order
    if $l < r$ **then**
        $p \leftarrow \mathit{Partition}(A[l..r])$ // p is index of pivot
        $\mathit{Quicksort}(A[l..p-1])$
        $\mathit{Quicksort}(A[p+1..r])$

Partition Algorithm

algorithm $\mathit{Partition}(A[l..r])$
    // Partitions a subarray using $A[l]$ as pivot
    // Input: Subarray of A
    // Output: Final position of pivot
    $\mathit{pivot} \leftarrow A[l]$
    $i \leftarrow l;\ j \leftarrow r+1$
    repeat
        repeat $i \leftarrow i+1$ **until** $A[i] \geq \mathit{pivot}$
        repeat $j \leftarrow j-1$ **until** $A[j] \leq \mathit{pivot}$
        if $i < j$ **then** swap $A[i]$ and $A[j]$
    until $i \geq j$
    swap $A[l]$ and $A[j]$
    return j

Illustration



Comments on Quicksort

- $\mathit{Partition}$ ensures that if p is the final position of the pivot, then $i < p$ implies $A[i] \leq A[p]$ and $i > p$ implies $A[i] \geq A[p]$.
- Best Case: If $\mathit{Partition}$ always splits subarray in half, then $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$.
- Worst Case: If pivot is always picks min or max, then $T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$.
- Average Case: Randomly chosen pivot is $\Theta(n \log n)$.
- In median-of-three partitioning, pivot is median of leftmost, rightmost, and middle element.
- Finding median is $\Theta(n)$ with a modification.

Binary Search Algorithm

algorithm $BinarySearch(A[0..n-1], K)$
  // Searches a sorted array using binary search
  // Input: An sorted array A and a key K
  // Output: The index of K or -1
  $l \leftarrow 0$; $r \leftarrow n-1$
  while $l \leq r$ **do**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ **then return** m
    else if $K < A[m]$ **then** $r \leftarrow m-1$
    else $l \leftarrow m+1$
  return -1

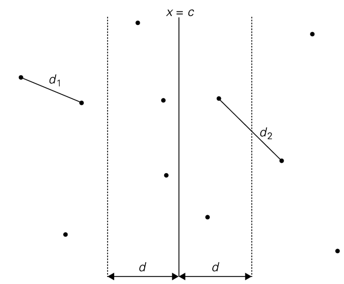
Illustration

K	31												
values in A	3	14	27	31	39	42	55	70	74	81	85	93	98
indices of A	0	1	2	3	4	5	6	7	8	9	10	11	12
iteration 1	*l*						*m*						*r*
iteration 2	*l*		*m*			*r*							
iteration 3				*l*	*m*	*r*							
iteration 4				*l,m,r*									

Comments on Binary Search

- $BinarySearch$ ensures that $A[l-1] < K$ (or $l = 0$) and $K < A[r+1]$ (or $r = n-1$).
- The number of possible indices is $r - l + 1$, which reduces by half or more each iteration.
- The number of times n can divided by 2 until it becomes < 1 is about $\log_2 n$ (exact $1 + \lfloor \log_2 n \rfloor$)

Idea for Closest Pair



Closest Pair Algorithm

algorithm $ClosestPair(P)$
  // Finds closest pair of points
  // Input: A set of n points sorted by coordinates
  // Output: Distance between closest pair
  if $n < 2$ **then return** ∞
  else if $n = 2$ **then return** distance between pair
  else
    $c \leftarrow$ median value for x coordinate
    $d_1 \leftarrow ClosestPair($points with $x < c)$
    $d_2 \leftarrow ClosestPair($points with $x > c)$
    $d \leftarrow \min(d_1, d_2)$
    $d_3 \leftarrow$ process points with $c - d < x < c + d$
    return $\min(d, d_3)$

Comments on Closest Pair

- Preprocessing can sort the points by their x and y coordinates (use two lists). Recursive calls can create sorted lists for subsets of points.
- The calculation of d_3 is $\Theta(n)$.
  - Select points with $c - d < x < c + d$.
  - Examine points in order of y values.
  - For any point (x, y), there can only be a few points within $(x \pm d, y \pm d)$.
- Preprocessing is $\Theta(n \log n)$, and recurrence is $T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$.

Convex Hull

First Idea for Convex Hull

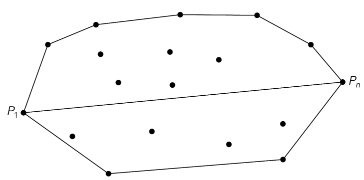


FIGURE 4.8 Upper and lower hulls of a set of points

Second Idea for Convex Hull

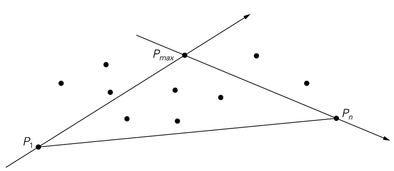


FIGURE 4.9 The idea of quickhull

Quickhull Algorithm

algorithm $Quickhull(P)$
  // Finds convex hull
  // Input: A set of n points
  // Output: A list of points, all of convex hull
  $P_1 \leftarrow$ point in P with lowest x value
  $P_2 \leftarrow$ point in P with highest x value
  $A \leftarrow$ points in P "above" $\overline{P_1 P_2}$
  $B \leftarrow$ points in P "below" $\overline{P_1 P_2}$
  return P_1, $QHWalk(P_1, P_2, A)$,
      P_2, and $QHWalk(P_2, P_1, B)$

Quickhull Algorithm Continued

algorithm $QHWalk(P_1, P_2, P)$
  // Finds convex hull clockwise from P_1 to P_2
  // Input: Two points and a set of n points
  // Output: A list of points, part of convex hull
  if $n \leq 1$ **then return** P
  else
    $P_3 \leftarrow$ point in P farthest from $\overline{P_1 P_2}$
    $A \leftarrow$ points in P "left" of $\overline{P_1 P_3}$
    $B \leftarrow$ points in P "left" of $\overline{P_3 P_2}$
    return $QHWalk(P_1, P_3, A)$, P_3,
      and $QHWalk(P_3, P_1, B)$

Comments on Quickhull

- $Quickhull$ and $QHWalk$ finds extreme points.
  - Points with lowest and highest x values are extreme.
  - Points farthest from any line are extreme.
  - P_3 is on the left of $\overline{P_1P_2}$ if $x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3 > 0$. Farthest point on left maximizes this value.
- Best Case: If $QHWalk$ always split points in thirds (A, B, and other), then $T(n) = 2T(n/3) + \Theta(n) \in \Theta(n)$.
- Worst Case: If split is bad, then $T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$.