Divide and Conquer

Hao Jiang
Boston College
Divide-and-Conquer Algorithms

- Break problems into subproblems
- Recursively solve each subproblem
- Combine the result
Merge Sort

Problem: Given an array of numbers, sort them into ascending order.

Idea: Instead of sorting the whole sequence directly, we partition the input into two halves and recursively sort each and combine.

\[
\text{sort}(A[1..n])
\]

\[
\text{sort}(A[1..\text{floor}(n/2)]) \quad \text{sort}(A[\text{ceil}(n/2)..n])
\]
Merge Sort

sort(A[1..n])

sort(A[1..floor(n/2)])

sort(A[ceil(n/2)..n])

combine
How to merge?

function merge(a[1..n], b[1..m])

    if n = 0 return b
    if m = 0 return a
    if a[1] < b[1]
        return [a[1], merge(a[2..n], b[1..m])]
    else
        return [b[1], merge(a[1..n], b[2..m])]
function mergesort(a[1...n])
Input: An array of numbers $a[1...n]$
Output: A sorted version of this array

if $n > 1$: 
    return merge(mergesort(a[1...[n/2]]), mergesort(a[[n/2] + 1...n]))
else:
    return a

Merge sort complexity: $T(n)=2T(n/2)+O(n)$
Node of nodes:

1

2

4

$2^\log(n) = n$

Problem size

n

n/2

n/4

Level k complexity = $O(n/(2^k)) \cdot 2^k = O(n)$

Total complexity = $O(n) \cdot \log n = O(n \log n)$
A divide and conquer method calls itself “a” times and each subproblem's is 1/b of the original one, and the complexity of merging result is \(O(n^d)\), then

\[ T(n) = aT(\text{ceil}(n/b)) + O(n^d) \]
The complexity

Size $n$

Size $n/b$

Size $n/b^2$

Branching factor $a$

Depth $\log_b n$

Width $a^{\log_b n} = n^{\log_b a}$
Master Theorem

If \( T(n) = aT(\lceil n/b \rceil) + O(n^d) \), \( a > 0 \), \( b > 1 \), \( d > 0 \) then

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a 
\end{cases}
\]
Order Statistics

Problem: Given an array of numbers, find the kth smallest number.

Example: 1 5 2 6 9

The 1st smallest is 1
The 3rd (the median) is 5
Median Filtering in Denoising

Original Image

Add 10% pepper noise
Median Filtering for Denoising

Median filter with 3x3 square structure element
Median Filtering for Denoising

Median filter with 5x5 square structure element
Compared with Gaussian Filtering

Kernel size 5x5 and sigma 3

Kernel size 11x11 and sigma 5
Divide and Conquer

Pick up a number \( v \) from the array, then reorganize the array into:

\[
\begin{align*}
&< v & = v & > v \\
\end{align*}
\]

This can be done in \( O(n) \) time.
Case 1: $k > p \land \land k < q$, the result is $v$

Case 2: $k \leq p$, we work on the subsequence $a[1..p]$ and find the $k$ th smallest number.

Case 3: $k \geq q$, we work on the subsequence $a[q..n]$ and find the $(k-q+1)$ th smallest number.
function select(a[1..n], k)

Choose an element v from a[1..n]
Organize a into three parts,

\begin{align*}
1 & \quad p & \quad q & \quad n \\
\end{align*}

\begin{align*}
< v & \quad =v & \quad > v \\
\end{align*}

if k > p and k < q, return v
if k <= q, return select(a[1..p], k)
if k >= q, return select(a[q..n], k-q+1)
The (worst case) Complexity

If we are very lucky, then each time the sequence is at least halved

\[ T(n) = T(n/2) + O(n) \implies T(n) = n \]

If we are very unlucky, each time the length of the sequence reduces by only 1

\[ T(n) = T(n-1) + O(n) \implies T(n) = 1 + 2 + 3 + \ldots + (n-1) = O(n^2) \]
The Average Complexity

We have $\frac{1}{2}$ chance to pick up a $v$ that is located in the range from 25% to 75%.

$$T(n) = T\left(\frac{3}{4}n\right) + \text{time to reduce the array from } n \text{ to } \frac{3}{4}n$$

The average complexity $T(n) = O(n)$
Quicksort

Quicksort picks up a pivot element $v$ and partition a sequence into two parts:

$$
\begin{array}{c}
\leq v \\
\geq v \\
\end{array}
$$

Then recursive sort the left and eight part
Quicksort

quicksort(A[1..n])
  if (n > 1)
    choose a pivot element A[p]
    k <- partition(A, p)
    quicksort(A[1..k-1]);
    quicksort(A[k+1..n]);
function partition(A[1..n], p)
    if (p not equal n)
    v ← A[n]
    i ← 1, j ← n-1
    while (i < j)
        while (i<j and A[i] <= v)
            i ← i + 1
        while (i<j and A[j] > v)
            j ← j - 1
        swap A[i] and A[j]
    swap A[i] with A[n]

Not processed
Quicksort Complexity

With good luck:
\[ T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n\log n) \]

With bad luck:
\[ T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2) \]

On the average:
\[ T(n) = O(n\log n) \]
Finding # of Reversed Pairs

Question: Given an array of distinct numbers, find how many number pairs are in the reversed order.

Example:

6 1 3 2 7

4 reversed pairs: (6,1), (6,3), (6,2), (3,2)
Divide and Conquer

\[
\text{count}(A[1..n])
\]

- \[
\text{count}(A[1..\text{floor}(n/2)])
\]
- \[
\text{count}(A[\text{ceil}(n/2)..n])
\]

\text{combine}
Divide and Conquer

\[ \text{count}(A[1..n]) \]

\[ \text{count}(A[1..\text{floor}(n/2)]) \quad \text{count}(A[\text{ceil}(n/2)..n]) \]

combine

But how to combine ???

We need a merging method this is faster than \( O(n) \)
Divide and Conquer

sort\_count(A[1..n])

\[
\text{sort\_count}(A[1..\text{floor}(n/2)]) \quad \text{sort\_count}(A[\text{ceil}(n/2)\ldots n])
\]

combine

Now there is a $O(n)$ merging method.
Linear Merging

Left Array

(normal order)

Merged result

Right Array
# of reversed pairs between left and right array = # of times we apply red arrow.
The algorithm

function sort_count(A[1..n])
    if n = 0
        return (A, 0)
    (L, n) = sort_count(A[1..floor(n/2)])
    (R,m) = sort_count(A[ceil(n/2), n])
    (B, k) = merge_count(L, R)
    return (B, n+m+k)
Matrix Multiplication

\[ Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}. \]

\[ T(n) = O(n^3) \]
Divide and Conquer

\[
\begin{array}{cccc}
A & B & \times & E & F \\
C & D & & G & H \\
\end{array}
\]

\[
\begin{array}{cc}
AE+BG & AF+BH \\
CE+DG & CF+DH \\
\end{array}
\]
Divide and Conquer

\[
\begin{array}{cc}
\begin{array}{c}
A \\
C
\end{array} & \begin{array}{c}
B \\
D
\end{array}
\end{array}
\times
\begin{array}{cc}
\begin{array}{c}
E \\
G
\end{array} & \begin{array}{c}
F \\
H
\end{array}
\end{array}
= \\
\begin{array}{cc}
\begin{array}{c}
AE+BG \\
CE+DG
\end{array} & \begin{array}{c}
AF+BH \\
CF+DH
\end{array}
\end{array}
\]

\[T(n) = 8T(n/2) + O(n^2)\]

\[\Rightarrow T(n) = T(n^3)\]
Can we do better?

\[ XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix} \]

\begin{align*}
P_1 &= A(F - H) \\
P_2 &= (A + B)H \\
P_3 &= (C + D)E \\
P_4 &= D(G - E) \\
P_5 &= (A + D)(E + H) \\
P_6 &= (B - D)(G + H) \\
P_7 &= (A - C)(E + F) \end{align*}

We only need 7 matrix multiplications in here.

\[ T(n) = 7T(n/2) + O(n^2) \Rightarrow T(n) = n^{\log_2(7)} \]
Finding Closest Point Pair

Which two points are the closest?
1D case
Divide and Conquer

Find \(d\) on the left
Find \(d\) on the right
Merge
Smart merge

No need to consider

No need to consider
Smart merge

Red dot to green dot distance must > d
Smart merge

There are no more than 4 blue dots!
There are no more than 6 blue dots!
The code

function get_smallest_d(Point p[1..n])
    sort points according to x and y coordinates
    and put into two arrays Px[1..n] and Ay[1..n]

    dleft = get_smallest_d(Px[1..floor(n/2)])
    dright = get_smallest_d(Px[ceil(n/2), n])
    d = min(dleft, dright)
    sequentially select point in Ay that falls
    in the band and form Py
    dy = get_smallest_7(Py)
    return min(d, dy)