Divide and Conquer

Maximum Contiguous Subarray Problem (MCS)

Divide: divide a given problem into two or more subproblems (ideally of approximately equal size)

Conquer: solving each subproblem (directly if small enough or recursively)

Combine: combining the solution of the subproblems into a global solution

DC Examples:

Maximum Contiguous Subarray Counting Inversions Integer Multiplication

Polynomial Multiplication Quicksort and Partition Deterministic and Randomized Selection

MCS Problem

Problem: Find the span of years in which a company earned the most

Input: An array of reals A[1…n]

The value of subarray A[i…j] is: $V(i,j) = \sum_{x=i}^{j} A(x)$

Find i⩽j, such that $V(i,j)$ is maximized.

A Brute Force Algorithm:

Calculate the value of $V(i,j)$ for each pair i⩽j and return the maximum value

Time Complexity: $O(n^3)$

A Data-reused Algorithm:

Don't need to calculate each $V(i,j)$ from scratch

Exploit the fact: $V(i,j) = \sum_{x=i}^{j} A[x] = V(i,j-1) + A[j]$

Time Complexity: $O(n^2)$

A divide-and-conquer Algorithm:

MCS = max(S1, S2, A)

S1: the MCS in A[1…m] S2: the MCS in A[m+1…n] A: the MCS across the cut

A = A1 ∪ A2

A1: MCS among contiguous subarray ending at A[m]

A2: MCS among contiguous subarray starting at A[m+1]

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

Time Complexity: O(n·logn)

Counting Inversions Problem

A Brute Force Algorithm:

List each pair $i < j$ and count the inversions

Time Complexity: $O(n^2)$

Divide-and-conquer Algorithm:

Divide: separate list into two halves A and B

Conquer: recursively count inversions in each list

Combine: count inversions (a, b) with $a \in A$ and $b \in B$

Return summation of above three counts

In the course of combination, assuming that A and B are sorted:

Scan A and B from left to right

Compare ai and bj,

If $a_i < b_j$, then ai is not inverted with any element left in B

If $a_i > b_j$, then bj is inverted with every element right in A

Function Merge-and-Count(A, B) can be executed in O(n) times where n is the number of elements in A and B

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

The Sort-and-Count algorithm counts the number of inversions in a permutation of size n in $O(n \cdot \log n)$ time

The Polynomial Multiplication Problem

What do we need to compute exactly?

$A(x) = \sum_{i=0}^{n} a_i x^i$

$B(x) = \sum_{i=0}^{m} b_i x^i$

$C(x) = A(x)B(x) = \sum_{k=0}^{n+m} c_k x^k$

Then

$c_k = \sum_{0 \le i \le n, 0 \le j \le m, i+j=k} a_i b_j$, for all $0 \le k \le n + m$

The vector C is the convolution of the vectors A and B (A major problem in digital signal processing)

Brute Force Algorithm:

Time Complexity: $O(n^2)$

The first Divide-and-Conquer Algorithm:

Divide: the original problem is divided into 4 problems of input size n/2

Conquer: compute $A_0(x)B_0(x), A_0(x)B_1(x), A_1(x)B_0(x), A_1(x)B_1(x)$ by recursively calling the algorithm 4 times

Combine: add the four polynomials O(n) times

T(n) = 4T(n/2) + n

Time Complexity: $O(n^2)$ Same order as the brute force algorithm

An improved divide-and-conquer algorithm:

What we really need are the following 3 terms:

$$A_0(x)B_0(x), \quad (A_0(x)B_1(x) + A_1(x)B_0(x)), \quad A_1(x)B_1(x)$$

The three terms can be obtained by using only three multiplications:

$$A_0(x)B_0(x), \quad A_1(x)B_1(x), \quad (A_0(x) + A_1(x))(B_0(x) + B_1(x))$$

T(n) = 3T(n/2) + n

Time Complexity: $O(n^{log3})$

Analysis of the D-C algorithm

The divide-and-conquer approach dose not always give you the best solution

There is actually an O(n·logn) solution to the polynomial multiplication problem

It involves using the Fast Fourier Transform algorithm as a subroutine

The FFT is another classic divide-and-conquer algorithm

The idea of using 3 multiplications instead of 4 is used in large-integer multiplications

Quicksort Problem

Partition:

Given: An array of numbers

Partition: Rearrange the array A[p…r] in place into two (possibly empty) subarrays A[p…q-1] and A[q+1…r] such that: $A[u] < A[q] < A[v]$ for any $p \leqslant u \leqslant q-1$ and $q+1 \leqslant v \leqslant r$

x is called the pivot

Use A[r] as the pivot, and grow partition from left to right

Initially, (i, j) = (p-1, p)

Increase j by 1 each time to find a place for A[j]

At the same time increase i when necessary

Stops when j = r

if A[j] > x: j += 1

if A[j] < x: i += 1, A[i] ←→A[j], j += 1

Time Complexity: O(r - p)

Randomized partition:

In the algorithm Randomized-Partition(A, p, r), we randomly choose an j, $p \leqslant j \leqslant r$, use A[j] as pivot

Idea is that if we choose randomly, then the chance that we get unlucky every time is extremely low

Average case analysis:

used for deterministic algorithms

assume the input is chosen randomly from same distribution

Expected case analysis:

used for randomized algorithms

need to work for any given input

Expected Running time of Randomized-Quicksort is $O(n \cdot \log n)$

Priority Queue:

Priority Queue is an abstract data structure that supports two operations:

- Insert: inserts the new element into the queue
- Extract-Min: removes and returns the smallest element from the queue

Possible implementations:

- Unsorted List:
  - Insert in O(1) time
  - Extract-Min in O(n) time
- Sorted Array:
  - Insert in O(n) time
  - Extract-Min in O(1) time
- Question: is there any data structure that supports both these operations in O(log n) time?

Binary Heap:

Heap-order Property: The value of a node is at least the value of its parent

If the heap-order property is maintained, heaps support the priority queue operation in O(log n) time

Insertion:

Add the new element to the next available position at the lowest level

Restore the min-heap property if violated

- General strategy is bubble up: if the parent of the element is larger than the element, then interchange the parent with its child

After each swap, the min-heap property is satisfied for the subtree rooted at the new element

Time Complexity = O(height) = O(log n)

Extract-Min:

Copy the last element to the root (overwrite the minimum element stored there)

Restore the min-heap property by bubble down: if the element is larger than either of its children, then interchange it with the smaller of its children

Time Complexity = O(height) = O(log n)

Heapsort

Build a binary heap of n elements

- The minimum element is at the top of the heap
- Insert n elements one by one: O(log n)

Perform n Extract-Min operations

- The elements are extracted in sorted order
- Each Extract-Min operation takes O(log n) time: O(n·log n)

Total Time Complexity: O(n·log n)

Heapsort is as efficient as merge sort and quicksort

Lower Bound for Comparison-based Sorting:

Objective:

All sorting algorithms seen so far are based on comparing elements

Insertion sort has worst-case running time $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \cdot \log n)$

Decision-Tree Model:

A decision tree can model the execution of any comparison-based sorting algorithm

Worst-case running time = height of tree

Lower Bound for Sorting:

Any comparison-based sorting algorithm require $\Omega(n \cdot \log n)$ comparisons

Proof:

A decision tree to sort n elements must have at least n! leaves, since there are n! possible orderings

A binary tree of height h has at most 2^h leaves

Thus, $n! \leqslant 2^h \Longrightarrow h \geqslant \log n! = \Omega(n \log n)$