CPE 231 Algorithms

Divide-and-Conquer

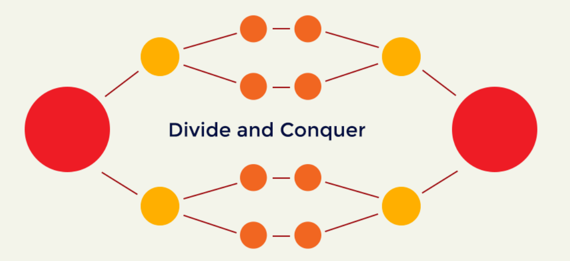
Dr. Taweechai Nuntawisuttiwong

Contents

Divide-and-Conquer

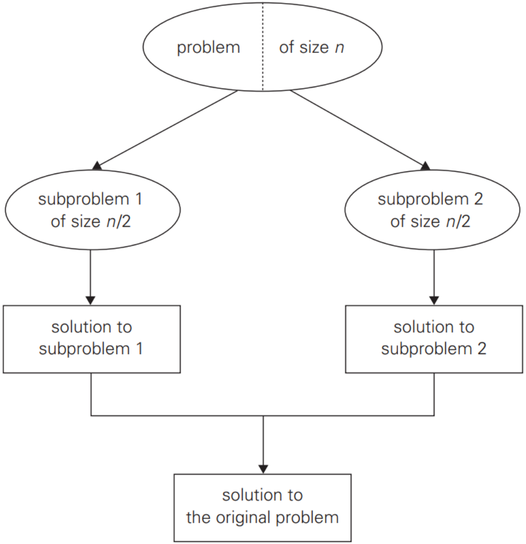
Introduction to Divide-and-Conquer

- Divide-and-conquer is a well-known algorithm design technique that efficiently solves problems by breaking them down into smaller subproblems.
- This method is widely used in computer science and has led to the development of many efficient algorithms.



Steps in Divide-and-Conquer

1. Divide the problem into several subproblems of the same type.
2. Solve the subproblems recursively.
3. Combine the solutions to get the final answer.



Example: Summing Numbers

Compute the sum of n numbers

we can divide the problem into two smaller sums:

1. Sum of the first half of the numbers.
2. Sum of the second half of the numbers.

This recursive approach continues until we reach a base case.

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

Efficiency of Divide-and-Conquer

- Not all divide-and-conquer algorithms are more efficient than brute-force solutions.
- However, many divide-and-conquer algorithms significantly reduce execution time.
- This technique is also suitable for parallel computations.

General Divide-and-Conquer Recurrence

The running time $T(n)$ of a divide-and-conquer algorithm can be expressed as:

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

where a is the number of subproblems,
b is the factor by which the problem size is reduced, and
$f(n)$ accounts for the time spent on dividing and combining.

Master Theorem

Definition

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence, then

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

Example: The summing numbers:

Mergesort

Mergesort

- Mergesort is a classic divide-and-conquer algorithm used for sorting.
- It divides an array into two halves, sorts each half recursively, and merges them back together in sorted order.

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)

Merging Process

- The merging process involves comparing elements from two sorted arrays.
- The smaller element is added to the new array, and the process continues until all elements are merged.

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 $A[k..p+q-1]$

Example

Sorting the list {8, 3, 2, 9, 7, 1, 5, 4}

Efficiency of Mergesort

The recurrence relation for the number of key comparisons:

The number of key comparisons in the merging stage:

Time complexity of mergesort:

Advantages and Disadvantages

Advantages

- Stable sorting algorithm (preserves the relative order of equal elements).
- Optimal for large datasets.

Disadvantages

- Requires additional space for temporary arrays.
- More complex to implement than other simple algorithms like QuickSort.

Quicksort

Quicksort

- Quicksort is a key sorting algorithm that utilizes the divide-and-conquer strategy. Unlike mergesort, it partitions elements based on their values rather than their positions.
- This method allows for efficient sorting by recursively sorting subarrays created from the partitioning process.

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

How Quicksort Works

1. Choose a pivot element from the array.
2. Partition the array into two subarrays—elements less than the pivot go to the left, and greater elements go to the right.
3. Recursively apply Quicksort to the subarrays.

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

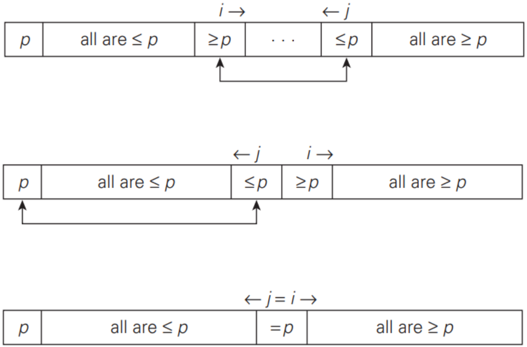
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow Partition(A[l..r])$ //s is a split position
  $Quicksort(A[l..s-1])$
  $Quicksort(A[s+1..r])$

Partitioning Process

- **Pivot Selection**: Typically, the first element of the subarray is chosen as the pivot.
- **Partitioning**: Elements are rearranged so that elements less than the pivot are on its left and greater elements on its right.
- **Hoare's Partitioning**: A more sophisticated method involving scanning from both ends of the array.

Three Situations After Scans Stop

1. Scanning indices have not crossed $(i < j)$
   - Swap $A[i]$ and $A[j]$
   - Resume scanning
2. Scanning indices have crossed $(i > j)$
   - Swap pivot with $A[j]$
3. Scanning indices stop at the same element $(i = j)$
   - Swap pivot with $A[j]$



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

Example of Quicksort

Sorting the list {5, 3, 1, 9, 8, 2, 4, 7}

Example of Quicksort

Tree of Recursive Calls to Quicksort

Efficiency of Quicksort

Best Case Scenario:

Occurs when the pivot divides the array into two nearly equal halves.

The number of key comparison:

Efficiency of Quicksort

Worst Case Scenario:

Occurs when the pivot results in highly unbalanced partitions, such as when the pivot is the smallest or largest element.

The number of key comparison:

Efficiency of Quicksort

Average Case Scenario:

Occurs with a randomly ordered array.

The number of key comparison:

Enhancements to Quicksort

- **Improved Pivot Selection**: Randomized Quicksort, Median-of-Three method.
- **Hybrid Approaches**: Use Insertion Sort for small subarrays or apply it at the end on nearly sorted arrays.
- **Three-way Partitioning**: Divides the array into elements less than, equal to, and greater than the pivot, improving performance on arrays with many duplicates.

Weaknesses of Quicksort

- Despite its advantages, Quicksort is not a stable sorting algorithm and requires additional stack space for recursive calls.
- Its performance can also be sensitive to the choice of pivot and the nature of the input data.

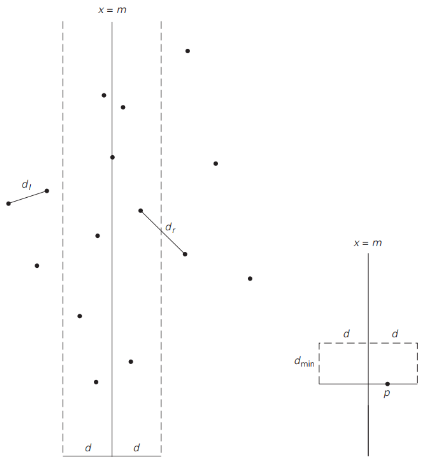
The Closest-Pair Problem by Divide-and-Conquer

Closest-Pair Problem

- Given a set of n points in the Cartesian plane, find the pair of points with the smallest Euclidean distance between them.
- The **brute-force method** solves the closest-pair problem in $O(n^2)$ time by checking all pairs of points. For $n \leq 3$, this method is straightforward and efficient.

Divide-and-Conquer Strategy

- Divide the set of points into two halves.
- Solve the problem recursively for each half.
- Combine the solutions by checking if there exists a closer pair across the dividing line.



Efficient Closest-Pair Algorithm

Recursive Approach:

Base Case:

- If the number of points $n \leq 3$, use the brute-force method.

Recursive Case:

- Calculate the minimum distance d_l in P_l and d_r in P_r.
- Set $d = \min(d_l, d_r)$.

Combining the Results:

- The closest pair might lie on opposite sides of the dividing line.
- Consider only points within a vertical strip of width $2d$ around the dividing line.

Efficient Closest-Pair Algorithm

ALGORITHM *EfficientClosestPair*(P, Q)

```
//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of n ≥ 2 points in the Cartesian plane sorted in
//       nondecreasing order of their x coordinates and an array Q of the
//       same points sorted in nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if n ≤ 3
    return the minimal distance found by the brute-force algorithm
else
    copy the first ⌈n/2⌉ points of P to array P_l
    copy the same ⌈n/2⌉ points from Q to array Q_l
    copy the remaining ⌊n/2⌋ points of P to array P_r
    copy the same ⌊n/2⌋ points from Q to array Q_r
    d_l ← EfficientClosestPair(P_l, Q_l)
    d_r ← EfficientClosestPair(P_r, Q_r)
    d ← min{d_l, d_r}
    m ← P[⌈n/2⌉ − 1].x
    copy all the points of Q for which |x − m| < d into array S[0..num − 1]
    dminsq ← d^2
    for i ← 0 to num − 2 do
        k ← i + 1
        while k ≤ num − 1 and (S[k].y − S[i].y)^2 < dminsq
            dminsq ← min((S[k].x − S[i].x)^2 + (S[k].y − S[i].y)^2, dminsq)
            k ← k + 1
return sqrt(dminsq)
```

Efficiency of the Algorithm

Time Complexity:

Recurrence Relation:

Solution: