Divide and Conquer Algorithms

This section covers two fundamental divide and conquer algorithms, Merge Sort and Quick Sort, and introduces a divide and conquer strategy for counting inversions in an array.

Merge Sort

Merge Sort divides an array into two halves, sorts each half, and then merges them back together.

```
[First Half] [Second Half]
```

Quick Sort

Quick Sort partitions an array around a pivot, then sorts the subarrays formed to the left and right of the pivot.

```
[<Pivot] [Pivot] [>Pivot]
```

Counting Inversions

An inversion in an array is a situation where two elements are out of their natural order, i.e., a pair `(i, j)` such that `i < j` and `A[i] > A[j]`.

Brute Force Method

The brute force method iterates through each element pair and counts the inversions, with a time complexity of $O(n^2)$.

```
cnt = 0;
for (i = 1 to n) {
    for (j = i + 1 to n) {
        if (A[i] > A[j]) {
            cnt++;
        }
    }
}
return cnt;
```

Divide and Conquer Strategy

By dividing the array and counting inversions within each part and across the parts, we can achieve more efficient counting, with a time complexity of $O(n \log n)$, similar to merge sort.

- Inversions within each part are counted during the sorting process.
- Cross inversions are counted when merging the two sorted halves.

The total inversions are the sum of left inversions, right inversions, and cross inversions. This is optimally done during the merge step of the Merge Sort algorithm.

Example for clarity:

```
Array A: [1, 4, 3, 5, 2]
Left Inversions: (4, 3)
Right Inversions: (5, 2)
Cross Inversions: Computed during merge step
```

The process of counting inversions can be incorporated into the merge step of the Merge Sort algorithm, allowing us to sort the array and count inversions simultaneously.

Divide and Conquer: Inversion Counting and Order Statistics

When both subarrays (left `L` and right `R`) are sorted, we can count cross inversions where an element in `L` is greater than an element in `R`.

For two sorted subarrays `L` and `R`:

L: [1, 2, 4, ...] R: [3, 5, 6, ...]

If `L[i] > R[j]`, all subsequent elements in `L` form inversions with `R[j]`.

To count cross inversions:

```
cross_inversion(L[1...n], R[1...m]) {
    z = 0;
    for i = 1 to n {
        for j = 1 to m {
            if (L[i] > R[j]) {
                z += (n - i + 1);
                break; // All subsequent elements in L will also be greater
            }
        }
    }
    return z;
}
```

Merge Sort and Inversion Counting

The function `count_inversion` counts the total number of inversions in an array by dividing it into halves and recursively counting inversions in each half and cross inversions.

```
count_inversion(A[1...n]) {
    if (n <= 1) return 0;
    X = count_inversion(A[1...n/2]);
    Y = count_inversion(A[n/2+1...n]);
    sort(A[1...n/2]);
    sort(A[n/2+1...n]);
    Z = cross_inversion(A[1...n/2], A[n/2+1...n]);
    return X + Y + Z;
}
```

Complexity Analysis

The recurrence for this divide and conquer approach is:

```
T(n) = 2 * T(n/2) + n * log(n)
```

An approximation for the upper bound is:

```
T(n) = 2 * T(n/2) + n * 1.01
```

Implying that $T(n) = O(n^{1.01})$, a slightly superlinear complexity.

Visual Representation of Recurrence

The recurrence for the divide and conquer approach can be visualized as a binary tree where each node represents a subproblem size and the cost of combining solutions.

```
          T(n)
          /  \
    T(n/2)    T(n/2)
    /  \        /  \
 T(n/4) T(n/4) T(n/4) T(n/4)
```

Each level of the tree represents a halving of the problem size, and the work done at each level is proportional to the number of subproblems multiplied by the cost to combine them.

Order Statistics

For example, to find the median, which is the 50th percentile, we can use a divide and conquer approach similar to Quick Sort, which is more efficient than sorting the entire array.

Divide & Conquer: Finding the k-th Smallest Element

Finding the k-th smallest element in an unsorted array is a classic problem known as order statistics.

```
A: [1, ... , n] k: n/2
```

Using a partition-based selection algorithm, similar to the one used in Quick Sort, we can find the k-th smallest element efficiently.

Algorithm Overview

Given an array `A` of `n` elements:

1. Partition the array around a pivot.
2. If the pivot's position `i` is equal to `k`, return `A[i]`.
3. If `i` is greater than `k`, recursively search the left subarray.
4. If `i` is less than `k`, recursively search the right subarray.

Pseudocode

```
kth(A[1...n], k) {
    j = Partition(A[1...n]);
    if (i == k) {
        return A[i];
    } else if (i > k) {
        return kth(A[1...i-1], k);
    } else {
        return kth(A[i+1...n], k - i);
    }
}
```

Complexity Analysis

- The average-case complexity is `O(n)` , but it can be worse in the worst case.
- Partition operation takes `O(n)` time.
- Recurrence: `T(n) = T(n/2) + n` , which simplifies to `O(n)` .

Visual Representation of the Partition Process

Initial array A: `[4, 1, 6, 2, 5, 3]`
Partitioning steps:
`[4, 1, 2] 3 [6, 5]` // Pivot = 3, k = 4
`[4, 5, 6]` // k = 1 (in the right subarray)
`[4]` // k = 1 (in the left subarray, which is the k-th element)

The process visualizes selecting the k-th smallest element by partitioning the array and reducing the problem size in each recursive call.

Recursive Tree Visualization for Complexity Analysis

```
    T(n)
     /  \
T(n/2) T(n/2)
```

k-th Smallest Using Heap

You can also use a Min-Heap or Max-Heap to find the k-th smallest or largest element, respectively.

Using Min-Heap for k-th Smallest

```
MinHeap(A[1...n]);
for (i = 1 to k-1) {
    MinHeap.deleteMin();
}
return MinHeap.getMin();
```

Using Max-Heap for k-th Largest

```
MaxHeap(A[1...n]);
for (i = n to k+1) {
    MaxHeap.deleteMax();
}
return MaxHeap.getMax();
```

Complexity for Heap Approach

- Building the heap is O(n).
- Deleting the k-1 smallest elements takes O(k log n).
- The overall complexity is O(n + k log n) for the k-th smallest.

Divide & Conquer II - Integer Multiplication

Recap

- Techniques such as counting inversions, finding the k-th smallest element, and sorting algorithms like Merge Sort and Quick Sort.

Integer Multiplication

- `int32` multiplication (max ~2-4 billion).
- `int64` multiplication (max ~4-16 quintillion).
- Application of RSA for bank security requires large integer multiplication of 2048 bits.

Example:

- Multiplying `X = 1234` and `Y = 1020` using the traditional method results in `T(n) = O(n^2)` for multiplication but only `T(n) = O(n)` for summation.

```
  1234
x 1020
------
  0000   (1234 * 0)
 0000    (1234 * 2, shift one position to the left)
1234     (1234 * 1, shift two positions to the left)
------
1258680  (Result of multiplication)
```

Using Divide & Conquer to Improve Multiplication

By splitting the digits and using recursive multiplication, we can improve the efficiency.

Example:

- `X = 1234` can be split into `12 * 10^2 + 34`.
- Given n digits numbers `X` and `Y`:

```
X = a * 10^(n/2) + b
Y = c * 10^(n/2) + d
```

Multiplication `X * Y` is:

```
(ac * 10^n) + ((ad + bc) * 10^(n/2)) + bd
```

- Using Divide & Conquer, we can reduce the problem into smaller subproblems.

Karatsuba Multiplication

To further reduce the number of recursive multiplications, we use the Karatsuba algorithm:

```
P1 = a * c
P2 = b * d
P3 = (a + b) * (c + d)
```

Then we can find the product by combining `P1`, `P2`, and `P3 - P1 - P2`.

Complexity Analysis

- The Karatsuba algorithm has a recurrence relation of `T(n) = 3T(n/2) + n` , leading to `T(n) = O(n^log2(3))` , which is approximately `O(n^1.58)` .

Visual Example for Karatsuba Multiplication

```
X = 1234  -> a = 12, b = 34
Y = 1020  -> c = 10, d = 20


P1 = 12 * 10 = 120
P2 = 34 * 20 = 680
P3 = (12 + 34) * (10 + 20) = 46 * 30 = 1380


Result = (P1 * 10^n) + (P3 - P1 - P2) * 10^(n/2) + P2
        = 120 * 10^4 + (1380 - 120 - 680) * 10^2 + 680
        = 1258680
```

Pseudocode for Karatsuba Algorithm

```
KaratsubaMultiplication(X, Y) {
    if (X < 10 or Y < 10)
        return X * Y

    /* Calculate the size of the numbers */
    m = max(size_base10(X), size_base10(Y))
    m2 = m / 2

    /* Split the digit sequences at the middle */
    high1, low1 = split_at(X, m2)
    high2, low2 = split_at(Y, m2)

    /* 3 calls made to numbers approximately half the size */
    z0 = KaratsubaMultiplication(low1, low2)
    z1 = KaratsubaMultiplication((low1 + high1), (low2 + high2))
    z2 = KaratsubaMultiplication(high1, high2)

    return (z2 * 10^(2 * m2)) + ((z1 - z2 - z0) * 10^(m2)) + z0
}
```

Max Sum of Sub-array

Find the maximum sum of any sub-array within a given integer array `A[1...n]`.

Examples

- For the array `[2, -1, 3, 5, -10, 4]`, the max sub-array sum is `9` (from the sub-array `[2, -1, 3, 5]`).
- Another example using Divide & Conquer:
  The ideal method is to start from the middle and look for numbers by left and by right.
  imagine the following array is from left to middle (n/2).

  ```
  index:[ 1,  2, 3, 4,  5, 6, ...]
  array:[-10, 7, 3, 2, -6, 5, ...]
  max-left-sum = 11 (from the sub-array [7, 3, 2, -6, 5])
  ```

Pseudocode

```
max_sum(A[1...n]) {
  // Base case: if the array has only one element
  if (n == 1) {
    return A[1]; // or max(A[1], 0) if we want to account for negative-only arrays
  }
  max_left_sum = find_max_sum(A[1...n/2])
  max_right_sum = find_max_sum(A[n/2+1...n])
  max_cross_sum = find_max_cross_sum(A, n/2)
  return max(max_left_sum, max_right_sum, max_cross_sum)
}

find_max_cross_sum(A, mid) {
  left_sum = 0
  left_max = 0
  for i = mid downto 1 {
    left_sum += A[i]
    if (left_sum > left_max) {
      left_max = left_sum
    }
  }
  right_sum = 0
  right_max = 0
  for i = mid+1 to n {
    right_sum += A[i]
    if (right_sum > right_max) {
      right_max = right_sum
    }
  }
  return left_max + right_max
}
```

Complexity Analysis

- For left and right side calculations, the time complexity is `T(n) = 2T(n/2) + n` , which simplifies to `T(n) = O(n log n)` .

Visual Representation

Array A = `[-10, 7, -2, 5, 4, -3, 6, 8, 1]`

Illustrating the divide & conquer approach:

- max_left = 10 (from sub-array [7, -2, 5])
- max_right = 6 (from sub-array [-3, 6, 8, 1])
- max_cross = 20 (crossing the middle, combining [7, -2, 5, 4, -3, 6, 8, 1])

Combination Step:

- Total sum = max(max_left, max_right, max_cross)

Combination

To find all subsets in an set A with n elements (e.g., n = 3), we can generate all possible subsets (2^n subsets):

All subsets of `A = {1, 2, 3}` :

```
{}, {1}, {2}, {3},
{1, 2}, {1, 3}, {2, 3},
{1, 2, 3}
```

For the algorithm, we can use a recursive approach to generate all these subsets.

Pseudocode for Combination

```
comb(A[1...n])
    if n == 0
        return [[]] // Base case: only the empty set

    sub = comb(A[1...n-1]) // Recurse with one less element
    solution = copy(sub) // Copy the subsets found so far

    // For each subset found, add the nth element and create new subsets
    for each s in sub
        new_subset = s + [A[n]]
        solution.append(new_subset)

    return solution
```

Permutations

Concept

- The number of permutations of a set of `n` elements is `n!` (n factorial).
- The algorithm for generating permutations involves swapping elements and recursing on the sub-array.

Pseudo Code

```
def perm(A):
    if len(A) == 1:
        return [A]
    solution = []
    for i in range(len(A)):
        A[i], A[-1] = A[-1], A[i]  # Swap ith and last element
        sub = perm(A[:-1])          # Generate permutations of the sub-array
        for s in sub:
            solution.append(s + [A[-1]])
        A[i], A[-1] = A[-1], A[i]  # Swap back for the next iteration
    return solution
```

Example

For `A = {1, 2, 3}`, the permutations are:

```
[
  [1, 2, 3], [1, 3, 2], [2, 1, 3],
  [2, 3, 1], [3, 1, 2], [3, 2, 1]
]
```

8 Queens Problem

- The 8 Queens puzzle asks to place 8 queens on an 8x8 chessboard so no two queens attack each other.
- A solution can be represented as a permutation of column positions where each number represents the column position of a queen in the corresponding row.

Pseudo Code

```
def solve_n_queens(n):
    # Assume a function `is_valid` that checks if the queen's position is valid
    # Assume a function `perm` that generates all permutations of an array
    for solution in perm(list(range(1, n + 1))):
        if is_valid(solution):
            return solution
    return []
```

Visual Example

```
[2, 4, 6, 8, 3, 1, 7, 5]  # Each number corresponds to the queen's column in that row
```

Binary Search Tree & Hashing

- Binary Search Trees (BSTs) and Hashing are two methods to maintain a dynamic set of items.
- BSTs offer `O(log n)` performance for insertion, deletion, and find operations, while hashing can offer `O(1)` performance on average.

Comparison Table

Operation	Binary Search Tree	Hashing
Insert	O(log n)	O(1)
Delete	O(log n)	O(1)
Find	O(log n)	O(1)

Data Structures

- **Set**: Can be implemented as a `TreeSet` (based on BST) or `HashSet` (based on hashing).
- **Map**: Can be implemented as a `TreeMap` (based on BST) or `HashMap` (based on hashing).

Example Usage

```
# TreeSet example in Python using `set`
tree_set = set()
tree_set.add(1)
tree_set.remove(1)

# HashMap example in Python using `dict`
hash_map = {}
hash_map['key'] = 'value'
```

Binary Search Tree (BST)

A binary search tree is a data structure that maintains a sorted order of elements, allowing for fast lookup, addition, and removal of items.

Visual Example:

```
    5
   / \
  3   7
 / \   \
1   4   8
```

- Here `5` is the root of the tree.
- All elements in the left subtree of `5` are less than `5`.
- All elements in the right subtree of `5` are greater than `5`.

Operations:

Search Operation

```
find(TreeNode root, int k) {
    if (root == null) return false;
    if (root.key == k) return true;
    if (k < root.key) return find(root.left, k);
    else return find(root.right, k);
}
```

- Worst-case time complexity: `O(h)` where `h` is the height of the tree.
- Average-case time complexity: `O(log n)` if the tree is balanced.

Insertion Operation

```
insert(TreeNode root, int k) {
    if (root == null) return new TreeNode(k);
    if (root.key == k) return root;
    if (k < root.key) root.left = insert(root.left, k);
    else root.right = insert(root.right, k);
    return root;
}
```

- Worst-case time complexity: `O(h)` where `h` is the height of the tree.

Deletion Operation

1. If the node to be deleted is a leaf, it can be removed directly.
2. If the node to be deleted has one child, remove the node and link its child with the node's parent.
3. If the node to be deleted has two children, find the in-order successor of the node.
    - The in-order successor is the smallest node in the right subtree of the node to be deleted.
    - Swap the values of the node and its in-order successor and then remove the in-order successor, which will now have at most one child.

```
delete(TreeNode* root, int k) {
    if (root == NULL) return root; // If the tree is empty or node not found
    if (k < root->key) {
        root->left = delete(root->left, k); // Node to be deleted is in the left subtree
    } else if (k > root->key) {
        root->right = delete(root->right, k); // Node to be deleted is in the right subtree
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            TreeNode* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            TreeNode* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children, get the inorder successor (smallest in the right subtree)
        TreeNode* temp = minValueNode(root->right);
        // Copy the inorder successor's content to this node
        root->key = temp->key;
        // Delete the inorder successor
        root->right = delete(root->right, temp->key);
    }
    return root; // Return the updated tree root
}

TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    /* Loop down to find the leftmost leaf */
    while (current && current->left != NULL)
        current = current->left;
    return current;
}
```

- Worst-case time complexity: `O(h)` where `h` is the height of the tree.

Finding In-Order Successor

```
successor(TreeNode root) {
    TreeNode t = root.right;
    while (t.left != null) t = t.left;
    return t;
}
```

- This function returns the in-order successor of a node in the BST, which is the node with the smallest key greater than the node's key.

In-Order Traversal Example:

```
[2, 5, 6, 7, 11]
```

- This sequence represents the keys of a BST when traversed in-order.