Advanced Algorithms Tutorial (C Language)

This tutorial covers several advanced data structures and algorithms that are commonly used in competitive programming and high-performance systems. Each section includes a clear explanation and a C implementation.

Tries

A trie (prefix tree) is a specialized tree structure for storing strings or associative arrays with string keys. Each node represents a character, and paths from the root to leaves form complete words. Key features:

- **Prefix-based organization**: Strings sharing common prefixes share nodes, enabling efficient prefix searches (e.g., autocomplete).
- **Operations**: Insertion, deletion, and search occur in $O(L)$ time, where L is the string length.
- **Variants**: Standard tries, compressed tries (space-optimized), and suffix tries (for substring searches).
  Applications include spell checkers, IP routing tables, and genomic sequence analysis[1][2][3].

Segment Tree with Lazy Propagation

A segment tree efficiently handles **range queries** (e.g., sum/min/max over $[l,r]$) and **range updates** on arrays. Lazy propagation optimizes updates:

- **Postponed updates**: Changes are temporarily stored in "lazy" nodes and applied only when needed.
- **Efficiency**: Range updates and queries run in $O(\log n)$ time.
- **Use cases**: Dynamic range minimum/maximum queries, interval scheduling with updates[4].

Suffix Array

A suffix array is a sorted array of all suffixes of a string. Unlike suffix trees, it uses less memory while supporting similar operations:

- **Construction**: Sorts all suffixes (e.g., using doubling algorithm).
- **Applications**: Pattern matching in $O(m \log n)$ time, longest common prefix (LCP) computation, and text compression.
  Suffix arrays are widely used in bioinformatics for DNA sequence alignment[5].

AVL/Red-Black Trees

Self-balancing BSTs that maintain height balance for efficient operations:

- **AVL Tree**:
    - Strict balance: Heights of left/right subtrees differ by ≤1.
    - Rotations rebalance after insertions/deletions.
- **Red-Black Tree**:
    - Less strict: Uses color rules (red nodes have black children; equal black nodes on all paths).
    - Guarantees $O(\log n)$ search/insert/delete.
    Both are used in database indices and language libraries (e.g., Java's `TreeMap`)[6][7].

Persistent Segment Tree

A variant of segment trees that preserves historical versions after updates:

- **Immutability**: New versions share unchanged nodes with prior versions.
- **Efficiency**: Updates use $O(\log n)$ space per change.
- **Applications**: Time-travel queries (e.g., "array state at time t"), historical data analysis[8].

Heavy-Light Decomposition (HLD)

HLD decomposes a tree into disjoint "heavy" and "light" paths for efficient path queries:

- **Heavy edges**: Connect nodes to their largest subtrees.
- **Light edges**: All other edges.
- **Queries**: Path operations (e.g., max/min) run in $O(\log^2 n)$ time.
  Used in network routing and tree-based dynamic programming[4].

Mo's Algorithm

An offline algorithm for **range queries** on arrays:

- **Block processing**: Divides array into n-sized blocks.
- **Query reordering**: Sorts queries by block, then by right endpoint.
- **Efficiency**: Answers Q queries in $O((n+Q)n)$ time.
  Ideal for problems where sorting queries minimizes redundant work (e.g., distinct element counts)[4].

Centroid Decomposition

Recursively partitions a tree into subtrees using **centroids** (nodes whose removal minimizes subtree size):

- **Centroid property**: No subtree exceeds $n/2$ nodes.
- **Depth**: The decomposition tree has $O(\log n)$ depth.
- **Applications**: Fast path queries (e.g., proximity detection) and tree isomorphism testing[4].

Link/Cut Trees

Dynamic data structures for representing forests of trees:

- **Operations**: Supports `link` (add edge), `cut` (remove edge), and path queries (e.g., max edge weight).
- **Splay trees**: Underlying mechanism for amortized $O(\log n)O(\log n)$ operations.
- **Use cases**: Network connectivity, dynamic graph algorithms, and robotics path planning[4].

Each structure optimizes specific computational challenges—from string processing (tries, suffix arrays) to dynamic data maintenance (segment trees, self-balancing BSTs) and tree path queries (HLD, centroid decomposition).

1. Tries (Prefix Tree)

Efficient for storing and querying strings, especially for autocomplete and dictionary matching.

C Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define ALPHABET_SIZE 26

struct TrieNode {
    struct TrieNode *children[ALPHABET_SIZE];
    bool isEndOfWord;
};

struct TrieNode *getNode(void) {
    struct TrieNode *pNode = (struct TrieNode *)malloc(sizeof(struct
TrieNode));
    pNode->isEndOfWord = false;
    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;
    return pNode;
}

void insert(struct TrieNode *root, const char *key) {
    struct TrieNode *pCrawl = root;
    for (int level = 0; key[level] != '\0'; level++) {
        int index = key[level] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();
```

```
        pCrawl = pCrawl->children[index];
    }
    pCrawl->isEndOfWord = true;
}
```

---

2. Segment Tree with Lazy Propagation

Efficient range updates and queries.

C Code:

```
#include <stdio.h>
#define MAX 1000
int seg[4*MAX], lazy[4*MAX];

void updateRange(int node, int start, int end, int l, int r, int val) {
    if (lazy[node] != 0) {
        seg[node] += (end - start + 1) * lazy[node];
        if (start != end) {
            lazy[2*node] += lazy[node];
            lazy[2*node+1] += lazy[node];
        }
        lazy[node] = 0;
    }

    if (start > end || start > r || end < l) return;

    if (start >= l && end <= r) {
        seg[node] += (end - start + 1) * val;
        if (start != end) {
            lazy[2*node] += val;
            lazy[2*node+1] += val;
        }
        return;
    }

    int mid = (start + end) / 2;
    updateRange(2*node, start, mid, l, r, val);
    updateRange(2*node+1, mid+1, end, l, r, val);
    seg[node] = seg[2*node] + seg[2*node+1];
}
```

---

3. Suffix Array

Used in full-text indices, data compression, and bioinformatics.

C Code (Naive):

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int cmp(const void *a, const void *b) {
    return strcmp(*(const char **)a, *(const char **)b);
}

void buildSuffixArray(char *txt, int n) {
    char *suffixes[n];
    for (int i = 0; i < n; i++)
        suffixes[i] = txt + i;

    qsort(suffixes, n, sizeof(char *), cmp);

    printf("Suffix Array:\n");
    for (int i = 0; i < n; i++)
        printf("%ld\n", suffixes[i] - txt);
}
```

---

4. AVL / Red-Black Tree

Self-balancing BSTs. We'll show AVL tree insertion here.

C Code (AVL Insert):

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int key, height;
    struct Node *left, *right;
};

int height(struct Node *N) {
    return N ? N->height : 0;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
```

```c
    node->left = node->right = NULL;
    node->height = 1;
    return node;
}

struct Node* rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}
```

---

5. Persistent Segment Tree (Versioned Updates)

Useful for range queries with rollback/history.

C Code (Basic Idea):

```c
// Omitted for brevity, typically requires dynamic memory allocation
// and immutability. Each version points to updated nodes.
```

---

6. Heavy-Light Decomposition (HLD)

Used to answer path queries on trees efficiently.

Idea: 1. Decompose tree into chains. 2. Use segment trees on chains for queries.

Code Tip: HLD is complex; requires DFS + Segment Tree combination.

---

7. Mo's Algorithm

Answer offline range queries efficiently.

C Code (Conceptual):

```c
// Sort queries by block and right endpoint.
// Use add/remove operations as you shift window.
```

---

8. Centroid Decomposition

Decomposes tree into centroids for divide-and-conquer algorithms.

Used for: Distance queries, path count with properties.

---

9. Link-Cut Trees

Advanced dynamic trees supporting path queries and updates.

Implemented using: Splay Trees.

Note: Too large for one snippet. Usually implemented with parent, path-parent pointers.

This document introduces advanced algorithms that are essential in solving large-scale problems involving trees, sequences, and queries.

Tries
https://www.youtube.com/watch?v=3CbFFVHQrk4
Segment Tree with Lazy Propagation
https://www.youtube.com/watch?v=ZBHKZF5w4YU
Suffix Array/Tree
https://www.youtube.com/watch?v=oSkWLYK6Huc
AVL Tree
https://www.youtube.com/watch?v=L0cDtyphZRc
Red Black Tree
https://www.youtube.com/watch?v=UaLIHuR1t8Q
Persistent Segment Tree
https://www.youtube.com/watch?v=O7_w02b7V8k
Heavy Light Decomposition
https://www.youtube.com/watch?v=4xjdsGkqhzE
Mo's Algorithm
https://www.youtube.com/watch?v=REI5E6Lr1Tk
Centroid Decomposition
https://www.youtube.com/watch?v=G8Q9pU6U19A
Link/Cut Trees
https://www.youtube.com/watch?v=YBSt1jYwVfU