Team notebook

8 de noviembre de 2017

Índice

1. Data Structures

1.1. Dynamic Segment Tree

```
/* Segment tree for sum in a very large range (LIM=10^9) */
struct node {
    int l, r, v;
    node() {l = r = -1, v = 0;}
};
vector<node> segtree;

/* Initial build with 0 for all items */
void build() {
    segtree.push_back(node());
}

/* Query the sum in a range a...b */
int query(int i, int l, int r, int a, int b) {
    if (i == -1 || b < l || a > r) {
        return 0;
    } else if (a <= l && r <= b) {
        return segtree[i].v;
    } else {
        int v1 = query(segtree[i].l, l, (l+r)/2, a, b);
        int v2 = query(segtree[i].r, (l+r)/2+1, r, a, b);
        return v1+v2;
    }
}

int query(int l, int r) {return query(0, 1, LIM, l, r);}

/* Add a node for complement the traversing path */
int add_node() {
    segtree.push_back(node());
    return segtree.size()-1;
}

/* Add v to item in position p */
void update(int i, int l, int r, int p, int v) {
    if (l == r) {
        segtree[i].v += v;
    } else {
        if (p <= (l+r)/2) {
            if (segtree[i].l == -1) segtree[i].l = add_node();
            update(segtree[i].l, l, (l+r)/2, p, v);
        } else {
            if (segtree[i].r == -1) segtree[i].r = add_node();
            update(segtree[i].r, (l+r)/2+1, r, p, v);
```

```
        }
        int v1 = (segtree[i].l == -1) ? 0 : segtree[segtree[i].l].v;
        int v2 = (segtree[i].r == -1) ? 0 : segtree[segtree[i].r].v;
        segtree[i].v = v1+v2;
    }
}

void update(int p, int v) {update(0, 1, LIM, p, v);}
```

1.2. Lazy Segment Tree

```
/* Segment tree for swapping & counting items color in range */
struct node {
        int lazy, black, white;
        node() {lazy = black = white = 0;}
} segtree[3*SIZE];

/* Initial build, all values are white */
void build(int i, int l, int r) {
    if (l == r) {
        segtree[i].white = 1;
    } else {
        build(2*i, l, (l+r)/2);
        build(2*i+1, (l+r)/2+1, r);
        segtree[i].white = segtree[2*i].white+segtree[2*i+1].white;
    }
}

/* Lazy propagation function for visited nodes */
void lazyprop(int i, int l, int r) {
    if (l < r) {
        segtree[2*i].lazy += segtree[i].lazy;
        segtree[2*i+1].lazy += segtree[i].lazy;
    }
    if (segtree[i].lazy%2 != 0) {
        swap(segtree[i].white, segtree[i].black);
    }
    segtree[i].lazy = 0;
}

/* Query the # of black items in a range a...b */
int query(int i, int l, int r, int a, int b) {
    lazyprop(i, l, r);
    if (a <= l && r <= b) return segtree[i].black;
    if (b < l || a > r) return 0;
    int v1 = query(2*i, l, (l+r)/2, a, b);
    int v2 = query(2*i+1, (l+r)/2+1, r, a, b);
    return v1+v2;
}

int query(int l, int r) {return query(1, 1, SIZE, l, r);}

/* Swap all items color in a range a...b */
void update(int i, int l, int r, int a, int b) {
    lazyprop(i, l, r);
    if (a <= l && r <= b) {
        ++segtree[i].lazy;
        lazyprop(i, l, r);
    } else if (!(b < l || a > r)) {
        update(2*i, l, (l+r)/2, a, b);
        update(2*i+1, (l+r)/2+1, r, a, b);
        segtree[i].white = segtree[2*i].white+segtree[2*i+1].white;
        segtree[i].black = segtree[2*i].black+segtree[2*i+1].black;
    }
}

void update(int p, int v) {update(1, 1, SIZE, p, v);}
```

1.3. Persistent Segment Tree

```
/* Segment tree for kth order statistic in a range l...r */
struct node {
    int l, r, v;
    node() {l = r = -1, v = 0;}
    node(int _l, int _r, int _v) {l = _l, r = _r, v = _v;}
};
vector<node> segtree;
vector<int> roots, arr;

/* Initial build with 0 for all items */
int build(int l, int r) {
    if (l == r) {
        segtree.push_back(node());
        return segtree.size()-1;
    } else {
        int li = build(l, (l+r)/2);
```

```
        int ri = build((l+r)/2+1, r);
        segtree.push_back(node(li, ri, 0));
        return segtree.size()-1;
    }
}

/* Query the kth val if sorted in a range l...r */
int query(int l, int r, int a, int b, int k) {
    if (l == r) {
        return l;
    } else if (segtree[segtree[b].l].v-segtree[segtree[a].l].v >= k) {
        return query(l, (l+r)/2, segtree[a].l, segtree[b].l, k);
    } else {
        int nk = k-segtree[segtree[b].l].v+segtree[segtree[a].l].v;
        return query((l+r)/2+1, r, segtree[a].r, segtree[b].r, nk);
    }
}

int query(int l, int r, int k) {return query(1, LIM, roots[l-1],
    roots[r], k);}

/* Insert next item & add new nodes to segtree */
int insert(int l, int r, int i, int p) {
    if (l == r) {
        segtree.push_back(node(-1, -1, 1));
        return segtree.size()-1;
    } else {
        int li = segtree[i].l;
        int ri = segtree[i].r;
        if (p <= (l+r)/2) {
            li = insert(l, (l+r)/2, li, p);
        } else {
            ri = insert((l+r)/2+1, r, ri, p);
        }
        segtree.push_back(node(li, ri, segtree[li].v+segtree[ri].v));
        return segtree.size()-1;
    }
}

int insert(int i, int p) {return insert(1, LIM, roots[i], p);}

/* Initialize the segment tree & insert each arr item */
void init() {
    roots.push_back(build(1, LIM));
    for (int i = 0; i < arr.size(); ++i) {
        roots.push_back(insert(i, arr[i]));
    }
}
```

1.4. Segment Tree

```
/* Segment tree for RMQ */
int segtree[3*SIZE], arr[SIZE];

/* Initial build & assignment of values */
void build(int i, int l, int r) {
    if (l == r) {
        segtree[i] = arr[l-1];
    } else {
        build(2*i, l, (l+r)/2);
        build(2*i+1, (l+r)/2+1, r);
        segtree[i] = min(segtree[2*i], segtree[2*i+1]);
    }
}

/* Query the min in a range a...b */
int query(int i, int l, int r, int a, int b) {
    if (a <= l && r <= b) {
        return segtree[i];
    } else if (b < l || a > r) {
        return INF;
    } else {
        int v1 = query(2*i, l, (l+r)/2, a, b);
        int v2 = query(2*i+1, (l+r)/2+1, r, a, b);
        return min(v1, v2);
    }
}

int query(int l, int r) {return query(1, 1, SIZE, l, r);}

/* Set item in position p to have value v */
void update(int i, int l, int r, int p, int v) {
    if (l == r) {
        segtree[i] = v;
    } else {
        if (p <= (l+r)/2) {
            update(2*i, l, (l+r)/2, p, v);
        } else {
```

```
            update(2*i+1, (l+r)/2+1, r, p, v);
        }
        segtree[i] = min(segtree[2*i], segtree[2*i+1]);
    }
}

void update(int p, int v) {update(1, 1, SIZE, p, v);}
```

1.5. Union Find

```
/* Union Find container (negative is size) */
int uf[SIZE];

/* Find the component to which v belongs */
int find(int v) {return (uf[v] < 0) ? v : (uf[v] = find(uf[v]));}

/* Join components of u & v */
void join(int u, int v) {
    int pu = find(u), pv = find(v);
    if (uf[pu] > uf[pv]) swap(pu, pv);
    uf[pu] += uf[pv], uf[pv] = pu;
}

/* Initialize uf sizes (-1) */
void init() {memset(uf, 255, sizeof(uf));}
```

2. Strings

2.1. Aho Corasick

```
/* Node for english lowercase letters */
struct node {
    int edges[26], f;
    unordered_set<int> out;
    node() {
        f = 0;
        memset(edges, 255, sizeof(edges));
    }
};
vector<node> trie;

/* Given a word, add it to the trie */
void add(int curr, int pos, const string &s) {
    // Traverse the trie & add nodes when necessary
    for (int i = 0; i < s.size(); ++i) {
        if (trie[curr].edges[s[i]-'a'] == -1) {
            trie[curr].edges[s[i]-'a'] = trie.size();
            trie.push_back(node());
        }
        curr = trie[curr].edges[s[i]-'a'];
    }
    // Mark last node as string index
    trie[curr].out.insert(pos);
}

/* Build the aho-corasick automaton */
void ahocorasick() {
    queue<int> q;
    for (int i = 0; i < 26; ++i) {
        if (trie[0].edges[i] == -1) {
            trie[0].edges[i] = 0; // always fail to 0
        } else {
            q.push(trie[0].edges[i]);
        }
    }
    int u, v, w, s;
    unordered_set<int> out;
    unordered_set<int>::iterator it;
    while (q.size() > 0) {
        u = q.front(), q.pop();
        for (int i = 0; i < 26; ++i) {
            if (trie[u].edges[i] == -1) continue;
            // Perform KMP step like here
            v = trie[u].edges[i], w = trie[u].f;
            while (trie[w].edges[i] == -1) w = trie[w].f;
            trie[v].f = trie[w].edges[i];
            // Combine occurrences
            out = trie[trie[v].f].out;
            for (it = out.begin(); it != out.end(); ++it) {
                trie[v].out.insert(*it);
            }
            q.push(v);
        }
    }
}
```

```
/* Given a word, print occurrences of added patterns on it */
void search(const string &s) {
    vector<pair<int, int> > ans;
    unordered_set<int> out;
    unordered_set<int>::iterator it;
    for (int i = 0, curr = 0, j; i < s.size(); ++i) {
        while (trie[curr].edges[s[i]-'a'] == -1) {
            curr = trie[curr].f;
        }
        curr = trie[curr].edges[s[i]-'a'];
        out = trie[curr].out;
        for (it = out.begin(); it != out.end(); ++it) {
            ans.push_back(make_pair(i, *it));
        }
    }
    for (int i = 0; i < ans.size(); ++i) {
        cout << "{" << ans[i].first << ", ";
        cout << ans[i].second << "}" << '\n';
    }
}

/* Initialize the trie with the root node */
void init() {trie.push_back(node());}
```

2.2. Hashing

```
/* Given a string A and a target B, print the occurrences of B in A */
void hashing(const string &a, const string &b) {
    // Define X, MOD & mod function
    #define mod(n) (((n)%(MOD)+(MOD))%(MOD))
    const long long X = 137, MOD = 1000000007LL;

    // Build the hash function
    string s = b + "$" + a;
    int n = s.size();
    long long h[n+1], r[n+1];
    h[0] = 0, r[0] = 1;
    for (int i = 1; i <= n; ++i) {
        h[i] = mod(h[i-1]*X+s[i-1]);
        r[i] = mod(r[i-1]*X);
    }

    // Print occurrences
    int m = b.size();
    for (int i = m+1; i <= n; ++i) {
        if (mod(h[i]-mod(h[i-m]*r[m])) == h[m]) cout << i-2*m-1 << '\n';
    }
}
```

2.3. KMP

```
/* Given a string A and a target B, print the occurrences of B in A */
void kmp(const string &a, const string &b) {
    // Build the PI function
    string s = b + "$" + a;
    int n = s.size(), pi[n+1];
    pi[0] = pi[1] = 0;
    for (int i = 2, j; i <= n; ++i) {
        j = pi[i-1];
        while (j > 0 && s[j] != s[i-1]) j = pi[j];
        if (s[j] == s[i-1]) j++;
        pi[i] = j;
    }

    // Print occurrences
    int m = b.size();
    for (int i = 0; i <= n; ++i) {
        if (pi[i] == m) cout << i-2*m-1 << '\n';
    }
}
```

2.4. Suffix Array

```
/* Comparison function + vector for suffix array */
vector<long long> rank2;
bool cmp(const int i, const int j) {return rank2[i] < rank2[j];}

/* Calculate the suffix array for the given word */
vector<int> suffix_array(const string &s) {
    int n = s.size();
    vector<int> sa(n), rank(n);
    rank2.resize(n);
    // Initialize for iter 1
```

```
    for (int i = 0; i < n; i++) {
        sa[i] = i;
        rank[i] = s[i];
    }
    // Calculate for subsequent iters
    for (int len = 1; len < n; len *= 2) {
        for (int i = 0; i < n; i++) {
            rank2[i] = ((long long) rank[i]<<32)+((i+len < n) ?
                rank[i+len] : -1);
        }
        sort(sa.begin(), sa.end(), cmp);
        for (int i = 0; i < n; i++) {
            if (i > 0 && rank2[sa[i]] == rank2[sa[i-1]]) {
                rank[sa[i]] = rank[sa[i-1]];
            } else {
                rank[sa[i]] = i;
            }
        }
    }
    return sa;
}

/* Calculate the LCP array for the given word */
vector<int> lcp_array(const vector<int> &sa, const string &s) {
    int n = s.size();
    vector<int> rank(n);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    vector<int> ans(n);
    for (int i = 0, l = 0; i < n; i++) {
        if (rank[i] > 0) {
            int j = sa[rank[i]-1];
            while (s[i+l] == s[j+l]) l++;
            ans[rank[i]] = l > 0 ? l-- : l;
        }
    }
    return ans;
}
```

2.5. Trie

```
/* Node for english lowercase letters */
struct node {
    int edges[26];
    bool is_end;
    node() {
        is_end = false;
        memset(edges, 255, sizeof(edges));
    }
};
vector<node> trie;

/* Given a word, add it to the trie */
void add(int curr, const string &s) {
    // Traverse the trie & add nodes when necessary
    for (int i = 0; i < s.size(); ++i) {
        if (trie[curr].edges[s[i]-'a'] == -1) {
            trie[curr].edges[s[i]-'a'] = trie.size();
            trie.push_back(node());
        }
        curr = trie[curr].edges[s[i]-'a'];
    }
    // Mark last node as an end
    trie[curr].is_end = true;
}

/* Given a word, check if it exist in the trie */
bool exist(int curr, const string &s) {
    // Try to traverse the trie or return if node doesnt exist
    for (int i = 0; i < s.size(); ++i) {
        if (trie[curr].edges[s[i]-'a'] == -1) {
            return false;
        }
        curr = trie[curr].edges[s[i]-'a'];
    }
    // String exist if it's an end & not a preffix
    return trie[curr].is_end;
}

/* Initialize the trie with the root node */
void init() {trie.push_back(node());}
```

2.6. Z Algorithm

```
/* Given a string A and a target B, print the occurrences of B in A */
void zfunc(const string &a, const string &b) {
    // Build the Z function
```

```
    string s = b + "$" + a;
    int n = s.size(), z[n];
    z[0] = 0;
    for (int i = 1, l = 0, r = 0, k; i < n; ++i) {
        if (i > r) {
            l = r = i;
            while (r < n && s[r-l] == s[r]) ++r;
            z[i] = r-l, --r;
        } else {
            k = i-l;
            if (z[k] < r-i+1) {
                z[i] = z[k];
            } else {
                l = i;
                while (r < n && s[r-l] == s[r]) ++r;
                z[i] = r-l, --r;
            }
        }
    }

    // Print occurrences
    int m = b.size();
    for (int i = 0; i < n; ++i) {
        if (z[i] == m) cout << i-m-1 << '\n';
    }
}
```