Competitive Programming Hackpack

Tobias Pristupinn

March 16, 2020

Contents

1 Data Structures

1.1 UFDS

Construction: $O(n)$
Union Find with Path Compression: $\approx O(1)$
Union Find Data Structure, or Disjoint Set, uses an array ***id*** to keep track of the root of the set of each element. The following implementation is **0-indexed**.

```
class UnionFind {

    public:
        vector<int> id, size;
        int components;
        UnionFind(int n){
            for (int i = 0; i < n; i++){
                id.push_back(i);
                size.push_back(1);
            }
            components = n;
        }

        int find(int a){
            int root = a;
            while (id[root] != root) {
                root = id[root];
            }

            //Path compression
            while (a != root){
                int next = id[a];
                id[a] = root;
                a = next;
            }

            return root;
        }

        void unite(int a, int b){
            int root_a = find(a), root_b = find(b);
            if (root_a == root_b){
                return;
            }

            if (size[root_a] > size[root_b]){
                id[root_b] = root_a;
                size[root_a] += size[root_b];
            } else {
                id[root_a] = root_b;
                size[root_b] += size[root_a];
            }

            components--;
        }

```

```
        bool connected(int a, int b){ return find(a) == find(b);}

        int getSize(int a){return size[find(a)];}
};
```

1.2 Segment Tree

Construction: $O(n)$
Query: $O(\log n)$
Segment Tree is a data structure that can be used to quickly answer mutliple range queries on an array. It works by creating a binary tree, where each node represents a range of the array. The root node represents the full array, its children represent the left and right halves, and so on.

```
void build_segment_tree(int l, int r, int node=1){
    if (l == r){
        tree[node] = arr[l];
    } else {
        int mid = (l + r) / 2;
        build_segment_tree(l, mid, node*2);
        build_segment_tree(mid + 1, r, node*2+1);
        tree[node] = tree[node*2] + tree[node*2+1]; //For sum queries
    }
}

int sum_query(int i, int j, int l, int r, int node=1){
    //i, j is the range for the query. l, r are the bounds for the current node
    //i and j are both inclusive
    if (i <= l && j >= r){ //full coverage
        return tree[node];
    } else if (i > r || j < l) { //no coverage
        return 0; //0 because it shouldn't affect the sum query. If doing a max query, return -infinity
    }

    //partial coverage
    int mid = (l + r) / 2;
    int left = sum_query(i, j, l, mid, node*2);
    int right = sum_query(i, j, mid + 1, r, node*2+1);
    return left + right; //combine according to query type
}

vector<int> arr = {1, 3, -2, 8, -7};
vector<int> tree(arr.size()*4, -1);

int main(){
    build_segment_tree(0, arr.size() - 1);
    int i = 0, j = 2;
    cout << sum_query(i, j, 0, arr.size() - 1) << endl;
}
```

Listing 1: https://cp-algorithms.com/data_structures/segment_tree.html
https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/

2 Algorithms

2.1 Sieve of Erastothenes

Construction: $O(n \log \log n)$
Generates a vector of every prime number $\leq$ *limit*. The basic idea behind the Sieve of Eratosthenes is that at each iteration one prime number is picked up and all its multiples are eliminated. After the elimination process is complete, all the unmarked numbers that remain are prime.

```
vector<bool> sieve_erastothenes(unsigned ll limit){
    vector<bool> prime(limit, true);
    for (int p = 2; p*p <= limit; p++){
        if (prime[p]){
            for (int i = p*p; i <= limit; i += p){
                prime[i] = false;
            }
        }
    }

    return prime;
}
```

2.2 Number of Divisors

Returns the number of distinct divisors of a number using the sieve of erastothenes.

```
int num_divisors(ll n){
    vector<ll> primes = sieve_erastothenes(n);
    int total = 1;
    for (ll p : primes){
        int count = 0;
        while (n % p == 0){
            n = n / p;
            count++;
        }
        total *= (count + 1);
    }

    return total;
}
```

Listing 2: https://www.geeksforgeeks.org/total-number-divisors-given-number/

2.3 Fast Primality Check

```
bool is_prime(ll n) {
    if (n <= 1)  return false;
    if (n <= 3)  return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (ll i = 5; i * i <= n; i += 6)
        if (n % i == 0 || n % (i+2) == 0)
           return false;

    return true;
}
```

2.4 Generate All Subsets

Runtime: $O(2^n)$

```
vector<vector<int>> all_subsets(vector<int> vec){
    vector<vector<int>> subsets;
    int n = vec.size();
    // 1 << n is the same as 2**n
    for (int i = 0; i < (1 << n); i++){
        subsets.push_back(vector<int>());
        for (int j = 0; j < n; j++){
            if (i & (1 << j)){ //check if bit j is set
                subsets.back().push_back(vec[j]);
            }
        }
    }
    return subsets;
}
```

3 Misc

3.1 Program Structure

```
#include <bits/stdc++.h>
using namespace std;
#define ll long long

int main(){
  ios::sync_with_stdio(0), cin.tie(0); //Fast IO
}
```

3.2 Runtime Bounds

1 second : $\leq 10^8$ operations

5 seconds : $\leq 10^9$ operations

n	Bound
$\leq [10..11]$	$O(n!), O(n^6)$
≤ 22	$O(2^n)$
≤ 100	$O(n^4)$
≤ 400	$O(n^3)$
$\leq 10K$	$O(n^2)$
$\leq 1M$	$O(n \log_2 n)$
$\geq 1M$	$O(n)$

3.3 Bitmasks

Operation	Result
Right Shift >>	Shift bits n places to the right. Equivalent to dividing by 2^n
Left Shift <<	Shift bits n places to the left. Equivalent to multiplying by 2^n
NOT ~	Flips each bit
OR |	OR's each bit. Binary operator, requires two different numbers
AND &	AND's each bit. Binary operator, requires two different numbers
XOR ^	XOR's each bit. Binary operator, requires two different numbers

Note: Never do bitwise operations on a signed integer!

3.4 Math

Properties of Mod

- $(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$
- $(a * b) \bmod m = (a \bmod m * b \bmod m) \bmod m$

3.5 STL Data Structures

- **vector**: Random access, amortized $O(1)$ insertion at end
- **deque**: $O(1)$ insertion and deletion at ends, **has $O(1)$ random access also!**
- **stack/queue**: Provides an interface that wraps around deque
- **bitset**: Optimized array of booleans
- **map, unordered_map, set**
- **multiset**: Set with duplicates
- **priority_queue**: Max heap

3.6 STL Algorithms

- **Sorting:**
  - `sort`
  - `partial_sort`: Sorts a range $[i..j]$ as if sorting normally. However, elements not in $[i..j]$ are in unspecified order.
  - `stable_sort`: Maintain initial order of items that are equal
- **Searching:**
  - `lower_bound`: Iterator pointing to first element $\geq$ `val`. Must be sorted beforehand. `lower_bound(10) ->` $[1, \mathbf{10}, 10, 10, 23]$. $O(\log n)$ runtime.
  - `upper_bound`: Same as `lower_bound` but returns first element $>$ `val`. `upper_bound(10) ->` $[1, 10, 10, 10, \mathbf{23}]$. If an array is sorted, the distance between two elements can be calculated by `upper_bound - lower_bound` using pointer arithmetic.
  - `binary_search`:
- **Other:**
  - `next_permutation`: Generates next lexicographically greater permutation. Can be used to generate every ordering if array is sorted beforehand. `prev_permutation` also exists.
  - `partial_sum`: Generates prefix sum array

3.7 Debug a segfault

```
g++ -g program.cpp
gdb ./a.out
run
backtrace
```