Notes from nearly draining pursuits of mastery in Leetcode problems

Hardik Rajpal

March 22, 2023

Contents

1 STL

1.1 Sets and Maps

These (`set` and `map`) are ordered data structures; helpful when it is efficient to retain the ordering of a collection of elements during execution. Their unordered counterparts (`unordered_set` and `unorderd_map`) prioritize access time and maintain no order of their elements.

Declaration syntaxes

```
struct U{
    bool operator()(T t1, T t2)const{
        //logic comparing t1<t2.
    }
};
set<T,struct U> myset; multiset<T,struct U> mymulset;
map<T,V,struct U> mymap; multimap<T,struct U> mymulmap;
```

Listing 1: Sets and Maps

The ordering parameter `U` is crucial in cases where order between T is not inferable.

```
struct T{
    ...
    bool operator==(T t2)const{
        //return logic for t2==this.
    }
};
struct hashT{
    size_t operator()(T t)const{
        //std::hash<string or int or double>()(string or int or double)
        //return logic for hashing t. Use ^ << >> ~ | &
    }
};
unordered_set<T,struct hashT> uset;
unordered_map<T,V,struct hashT> umap;
```

Listing 2: Unordered sets and maps

1.2 Iterators

An iterator pointing at an element is "corrupted" on removing the element. Hence, any useful data should be copied over the iterator before removing the element.

```
    lists.erase(*minit);//minit is corrupted
    lists.insert((*minit)->next);
```

Listing 3: Undefined behaviour

```
    lists.insert((*minit)->next);//first use the data.
    lists.erase(*minit);//then erase.
```

Listing 4: Working code

1.3 Built-In Utilities

Note: `iter` denotes the iterator return type. `first` and `last` denote `.begin()` and `.end()` iterators.

- `void sort(first,last)` (O(nlogn))
- `void reverse(first,last)` (O(n))
- `void random_shuffle(first,last)` (O(n))

- `iter max_element(first,last[,struct comp])` (O(n))
- `iter min_element(first,last[,struct comp])` (O(n))
- `int | long long | etc accumulate(first,last,init_val)` (O(n))
- `iter lower_bound(first,last,value)`(O(logn))
  - Returns `iter` to the smallest element $\geq$ `value`.
- `iter upper_bound(first,last,value)` (O(logn))
  - Returns `iter` to the smallest element $>$ `value`.
- `bool next_permutation(first,last[, struct comp])` (O(n))
  - Updates `(first,last)` to its next permutation of in ascending order.
  - Sort `(first,last)` first to access all permutations.
  - Use in a `do-while` loop to avoid missing first permutation.
  - Returns true if there exists a permutation greater than the current one.
- `bool prev_permutation(first, last[, struct comp])` (O(n))

2 Graph Algorithms

2.1 BFS—DFS

I prefer writing both of these iteratively. In the immortal intonation of Ashish Mishra,
BFS - Queue
DFS - Stack
Here's a sample of both algorithms.

```
T s;
unordered_map<T,vector<T>> edges;
queue<T> q;
unordered_map<T,bool> visited;
unordered_map<T,T> prev;
int steps = 0;
q.push(s);
visited[s] = true;
while(!q.empty()){
    int sz = q.size();
    while(sz--){
        T u = q.front();
        q.pop();
        for(nb:edges[u]){
            if(!visited[nb]){
               visited[nb] = true;
               prev[nb] = u;
               q.push(nb);
            }
        }
    }
    steps++;
}

```

Listing 5: BFS

```
T s;
unordered_map<T,vector<T>> edges;
stack<T> s;
unordered_map<T,bool> visited;
unordered_map<T,T> prev;
s.push(s);
visited[s] = true;
while(!s.empty()){
    T u = s.top();
    s.pop();
    for(nb:edges[u]){
        if(!visited[nb]){
            visited[nb] = true;
            prev[nb] = u;
            q.push(nb);
        }
    }
}
```

Listing 6: DFS

The nodes should be reduced from the abstract type T, to int whenever possible; thereby reducing the `unordered_maps` to `vectors` which can sometimes get you under the time limit. Click here for Why?

Optimizations

- If the list of neighbours is a shared data structure, consider clearing it after having visited the neighbours using any one owner. Since, all elements in the shared field are visited and running them through the loop for other owners of the field is redundant. (Leetcode)

3 Misc. Algorithms

3.1 Binary Search

While the idea of binary search is clear, opportunities for its application may not be easily identified (yet). Some common places where it may be applied:

Optimization Problems

Problems involving the evaluation of the min/max of an expression, while its constituents satisfy a constraint that is straightforward to check. Consider the problem below:

Given $x_1, x_2, ...x_n$ and T, find $min_{a_1,a_2,...a_n}(max_i(a_i x_i))$ such that $\sum_{i=0}^{n} a_i \geq T$ (Leetcode)

The binary search algorithm is:

```
    int n = x.size();
    int mine = *min_element(x.begin(),x.end());
    int maxe = *max_element(x.begin(),x.end());
    unsigned long long lb = mine, ub = T*(unsigned long long)maxe;
    unsigned long long del = (ub - lb)/2;
    auto numtrips = [x,n](unsigned long long gt){
        unsigned long long nt = 0;
            for(int i=0;i<n;i++){
                nt += (gt/x[i]);
            }
            return nt;
    };
    while(del>0){
        if(numtrips(lb+del)>=T){
            ub = lb + del;
        }
        else{
            lb = lb + del;
        }
        del = (ub-lb)/2;
    }
    if(numtrips(lb)>=T){
        return lb;
    }
    return lb+1;
```

Listing 7: BinSearch

3.2 Array Scan

The name is given to the family of algorithms where we do a couple of runs of a given array to evaluate an attribute. Approaches involving subarrays can be dealt with using to two indices `s` and `e`. Things to note:

- Edge cases are possible at the start or end.
- The attribute evaluation will often be have to be done once more at the end of the loop.
- To improve performance, try to reduce the variables being updated/used in the loop.
- Two loops are useful for getting started with the code, but reducing them to one helps performance.