Notes from nearly draining pursuits of mastery in Leetcode problems

Hardik Rajpal

August 1, 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 ^ << >> ~ | &
        //Note: Easy hash for collection of numbers=> sort, join with ","
    }
};
unordered_set<T,struct hashT> uset;
unordered_map<T,V,struct hashT> umap;
```

Listing 2: Unordered sets and maps

In the interest of speed and space, and unreadability of code, it might be preferable to replace sets that only serve to mark and check membership by flags and bit operations.

`insert(index)≡flag|(1<<index)` and `count(index)≡flag&(1<<index)`

Note: The great thing about sets and maps is the uniqueness of their values. So, to find the maximum element less than an element, we simply `find()` the element in the set and decrement the iterator.

1.1.1 Erase

`erase` can take the value to be erased or the iterator to it. With multisets and multimaps, using the iterator ensures that the duplicates are not erased, whereas using the value erases all duplicates. Note that the iterator can't be a `reverse_iterator`; we can't use `rbegin()` and must use `end()` after decrementing it.

1.2 Priority Queue

These structures lazily maintain order between their elements; only one extreme of the elements in the data structure is accessible at any time. The implementation involves a heap. the usage is as below:

```
    struct U{
        bool operator()(T t1, T t2)const{
            //logic comparing t1<t2.
        }
```

```
    };
    priority_queue<T,vector<T>,struct U> pq;
```

Listing 3: Priority Queue

The default U is `std::less<T>`. `pq.top()` returns the largest element, based on the comparator.

1.2.1 Getting the Kth value

Some problems can ask for the Kth value (largest/smallest) from an array of `val`s. These can be implemented efficiently with PQs as follows:

```
    priority_queue<int,vector<int>,greater<int>> pq;
    //greater=> pq.top == least value of pq.
    pq.push(val[0]);
    for(int i=1;i<val.size();i++){
        if(pq.size()<k){
            pq.push(val[i]);
        }
        else if(pq.top() < val[i]){
            pq.push(val[i]);
            pq.pop();
        }
    }
    return pq.top();
    //size of pq == k, and we have
    //collected all high elements=>
    //pq.top == least value must be kth largest
```

Listing 4: Kth Largest Element

```
    priority_queue<int,vector<int>,less<int>> pq;
    //less=> pq.top == max value of pq.
    pq.push(val[0]);
    for(int i=1;i<val.size();i++){
        if(pq.size()<k){
            pq.push(val[i]);
        }
        else if(pq.top() > val[i]){
            pq.push(val[i]);
            pq.pop();
        }
    }
    return pq.top();
    //size of pq == k, and we have
    //collected all low elements=>
    //pq.top == max value must be kth smallest
```

Listing 5: Kth Smallest Element

The time complexity of these algorithms is O((`val.size()`)log(k)); it's linear in `val.size()`. An alternative way to maintain the kth element in a list of elements, where deletion of specific elements is necessary (but not permitted by the priority_queue data structure), we can use two sets.

```
class Kset{
public:
    size_t k;
    multiset<int> trail;//multiset to permit duplicates.
    multiset<int> others;
    Kset(size_t _k){
        k = _k;
    };
    void insert(int e){
        if(trail.size()<k){
            trail.insert(e);
        }
        else{
            if(e<*trail.rbegin()){
```

```
                others.insert(*trail.rbegin());
                auto iter = trail.end();iter--;
                trail.erase(iter);
                trail.insert(e);
            }
            else{
                others.insert(e);
             }
        }
    }
    void erase(int e){
        auto iter = others.find(e);
        if(iter!=others.end()){
            others.erase(iter);
            //erase using iterators to avoid
            //erasing all duplicates
        }
        else{
            iter = trail.find(e);
            if(iter!=trail.end()){
                trail.erase(iter);
                trail.insert(*others.begin());
                others.erase(others.begin());
            }
        }
    }
    int top(){
        //returns kth smallest element.
        return *trail.rbegin();
    }
}
```

Listing 6: Kth smallest Element

1.3 The Big Fat Table of Data Structures

STL class	Insertion	Deletion	Lookup	Remarks
`vector<T> s`	`void push_back(T t)`:O(1) `void insert(iter pos, T t)` O(n)	`pop_back`: O(1)	`var[key]`: O(1)	`insert` puts `t` before `pos`
`set<T> s`	`void insert(T t)`: O(log(s.size()))	`void erase(T t)`: O(log(s.size()))	`iter find(T t)`: O(log(s.size()))	Implemented as a tree.
`map<K,V>`	`void insert(pair<K,V> p)`: `var[k] = v;`	`void erase(K k)`: O(log(s.size()))	`iter find(K k)`: O(log(s.size()))	...
`priority_queue <T,vector<T>,U>`	`pq.push(T t)` O(log(`pq.size()`))	`pq.pop()` O(log(`pq.size()`))	`pq.top()` is O(1)	...

1.4 Comparators

STL provides its own simple comparators: (`T` can be replaced by int, vector, etc.)

- `less<T>` for ascending orders.
- `greater<T>` for descending orders.

Custom comparators are written like so:

```
    struct U{
        bool operator()(T t1, T t2)const{
            //logic comparing t1<t2.
        }
```

```
};
```

Listing 7: Comparators

1.5 Iterators

Without going into the non-trivial hierarchy of iterators, note that:

- `iter++` is supported by all iterators.
- `iter += n` is supported by random-access iterators, available with vectors and deques (and maybe others?).
- `void advance(iter,n)` is supported by all iterators: use this instead.
- `iter next(iter,n)` and `iter prev(iter,n)` is supported by all iterators.
- `distance(iter_before,iter_after)` is the more general version of `iter_before - iter_after`.

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

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

Listing 8: Undefined behaviour

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

Listing 9: Working code

1.6 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 10: 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 11: 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)

In such problems, the answer involves

- Drafting a binary search algorithm with `l, r` values chosen meaningfully.

- Drafting a `status(int m)` function that conveys which half of the search space we need to split.

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 12: BinSearch

3.2 Buckets

Splitting a linear range up into buckets of size $\sqrt{n}$, and maintaining necessary values (min, max, etc.) can help reduce time complexites from O(n) to O($\sqrt{n}$).

3.3 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.

3.4 Min/Max in Arrays

Some questions might require evaluating the min/max elements in subarrays. These can be pre-computed using **prefix** and **suffix** arrays. In a prefix array, `a[i]` denotes the min/max value of the set {`a[0],a[1]...a[i]`} and in a suffix array, it denotes the min/max value of the set {`a[i],a[i+1],...a[n-1]`}. Additionally, **prefix and suffix sum** arrays can be used to pre-compute cumulative sums for sub-arrays. If a question involves finding the max element less than or min element more than a value over a range, consider using **binary search** and maintaining a sorted version of the range.

3.4.1 Two constraints

Some problems reduce to finding for each element `e` in `arr2`, an element in `arr1` whose `attr1` is greater than a value `f(e)` and `attr2` is minimized. These problems can be solved in min(n1log(n1),n2log(n2)) as follows:

```
    sort(arr2.begin(), arr2.end(), [](auto &a, auto &b) {
        return f(a) > f(b);
    });
    sort(arr1.begin(), arr1.end(), [](auto &a, auto &b){
        return a.attr1 > b.attr1;
    });
    int i = 0;
    set<int> attr2ValSet;
    vector<int> ans(arr2.size());
    for(auto e : arr2) {
        int minAttr1 = f(e);
        while(i < arr1.size() && arr1[i].attr1 >= minAttr1){
            attr2ValSet.insert(arr1[i].attr2);
            i++;
        }
        if(st.size()) {
            auto it = st.begin();
            //minimal attr2 that satisfies attr1 >= f(e).
            ans[e.idx] = *it;
        }
        else{
            ans[e.idx] = -1;
        }
    }
```

Listing 13: Two constraints

3.4.2 Sub-Array Sum

Here are some ideas I find useful in subarray sum questions:

1. Compute the prefix—suffix sum arrays. Are they of any help?
2. Divide and conquer: Try checking the condition for
   (a) a[0]...a[n/2 - 1] (recursively)
   (b) a[n/2]...a[n - (n/2) - 1] (recursively)
   (c) Compute prefix sum of (a) and suffix sum of (b) and check for subarrays formed by combining the two.

   If (c) can be done in less than O(n^2) time, we've usually found a solution.
3. Consider a sliding window along the array to capture selected subarrays.

3.4.3 Spans

Questions that involve finding the latest previous element that is greater than the current value in a sequence can be solved better by ignoring any values that are surrounded by higher values; remove any element if it is smaller than its previous element.
TODO: get infographic?