Pragmatic Programming

Session 2 - Data Structures and Complexity

Max Nilsson

26th September

$\mathcal{O}$! Darling ...

- Formally: $f \in \mathcal{O}(g)$ if there exists M such that $f(x) \leq Mg(x)$ for x large enough $\iff \limsup f/g < +\infty$.
- With appropriate abuse of notation: $f(x) = \mathcal{O}(g(x))$. We will always use $\mathcal{O}$ in the tight sense (usually reserved for Ω).
  - `Binary Search` $= \mathcal{O}(\log n)$ is tight.
  - `Binary Search` $= \mathcal{O}(n)$ is true, but not tight.
  - This convention avoids contradictions such as

$$\mathcal{O}(n) = \texttt{Binary Search} = \mathcal{O}(\log n).$$

- Time complexity calculus works intuitively. For instance:

```
rep(i, 0, n) O(log n); // O(n log n)
```

- In Kattis, when writing in C++, if your code is correct and has good enough time complexity, it should pass (if the problem is not too difficult).
- In a simplified way, if your algorithm has time complexity $\mathcal{O}(f)$ and input size n, then if $f(n) \leq$ `1e8` you should be fine.

... Please Believe Me

- Avoid the common mistake when getting `Time Limit Exceeded` (`TLE`) of optimizing $\mathcal{O}(1)$-stuff (i.e., stuff that does not change the overall complexity). If your complexity is too large to begin with, this will never work.
- Consider the following example. You are given a list of $n \leq$ `1e6` integers and you want to know if there exists two elements that sum to some value x. Naive $\mathcal{O}(n^2)$ solution:

```
rep(i, 0, n) cin >> v[i];
rep(i, 0, n) rep(j, 0, n) if (v[i]+v[j] == x)
    {cout << "YES\n"; return; }
cout << "NO\n";
```

- If we replace the second line with

```
rep(i, 0, n) rep(j, i+1, n)
```

  we get something that is (twice) faster! But the time complexity is still $\mathcal{O}(n^2)$ and will get `TLE`.
- Important lesson: **you should always assume that the worst possible test case exists among the hidden ones.**

Example continued

- Amazingly, there exists two natural ways of solving the previous problem with a better complexity.
- First sort the vector `v`, with complexity $\mathcal{O}(n \log n)$, and then use a two-pointer method

```
int l = 0, u = n-1;
```

  which traverses the array with appropriate `l++` or `u--`, and complexity $\mathcal{O}(n)$. In total, we get complexity

$$\mathcal{O}(n \log n + n) = \mathcal{O}(n \log n).$$

- The fastest way is to simply linearly traverse the array, while keeping track of all the elements seen so far in an `unordered_set<int>`. Note that both `.insert(v[i])` and `.count(x-v[i])` have complexity $\mathcal{O}(1)$ and so the total complexity is linear $\mathcal{O}(n)$.

The Complexity of Factorization

- Given $n \leq$ 1e18 output the prime factors of n.
- Naive algorithm is searching through all $2 \leq k \leq \sqrt{n}$ and performing

```
while(!(n%k)) cout << k << '\n', n/=k;
```

  and if no k was outputted, then n is prime.
- This algorithm is $\mathcal{O}(\sqrt{n})$. But it should never be used!
- If you have 1e6 queries, each wanting the factors of some $n \leq$ 1e6 then do some preprocessing sieve. See Product Divisors.
- If you only have a few queries, but $n \leq$ 1e18 is large, then use Pollard's Rho Algorithm. It is a probabilistic algorithm with complexity $\mathcal{O}(n^{1/4})$.
- **Corpus Idea:** Improve the Omogen Heap implementation of Pollard.

Vector

- Dynamically allocated array, with append method `.pb(x)` with complexity dependent on memory availability. In general, you can treat it as an $\mathcal{O}(1)$-operation (at least according to Bjarne Stroustrup).
- The converse `.pop_back()` is definitely $\mathcal{O}(1)$.
- Consider using `v.back()` instead of the clumsy `v[v.size()-1]`.
- Lookup `v[i]` is constant but `.erase(v.begin()+i)` should be avoided as it is $\mathcal{O}(n)$.
- Iterate with either

```
rep(i, 0, v.size()) v[i];  or  for (auto x :  v);
```

  The keyword `auto` is modern `C++` where the compiler infers the type. It should not have any runtime downsides. Not writing `auto&` might have runtime downsides.
- Sort in $\mathcal{O}(n \log n)$

```
sort(all(v), [](auto p1, auto p2){[...]};
```

- The binary search methods, when `v` is sorted, `lower_bound(all(v), x)` and `upper_bound(all(v), x)` are often useful, and are $\mathcal{O}(\log n)$.

Stack and Queue

- Use `stack<int>` only for design purposes, since it is equivalent with `vector<int>`.
- Use `queue<int>` as a FIFO-structure, with useful methods `.push(x)`, `.front()`, `.pop()`, all with (1).
- Example: Given a graph `vi g[mxn];` which we know has a tree structure and 0 as a root node, compute the depth of each node.

```
int depth = 0;
queue<int> q = {0};
while(q.size()) {
  int x = q.size();
  while(x--) {
    int u = q.front(); q.pop();
    depths[u] = depth;
    for (int v :  g[u]) q.push(v);
  }
  depth++;}
```

Priority Queue

- Extremely useful heap structure. The simplest version is `priority_queue<int>` which is a **max** priority queue.
- If you want a **min** priority queue (such as in `Dijkstra's Algorithm`) you must can write

```
priority_queue<pii, vii, greater<pii>> pq;
```

  The `vii` means that the `priority_queue` should be implemented on top of the `vector` data structure, and `greater` gives us a min priority queue. Note that here `pii(d, v)` says that the distance between to `v` is `d`.
- The useful methods are `.push(x)`, `.top()`, `pop()`, which are $\mathcal{O}(\log n), \mathcal{O}(1), \mathcal{O}(\log n)$ respectively.

Priority Queue with own Comparator

- Unfortunately there seems to be no lambda-way of defining a `priority_queue` with your own custom comparator.
- I use the following:

```
class ComparisonClass {
public:
  bool operator() (const pii& p1, const pii& p2) {
  if (p1.first != p2.first) return p1.first < p2.first;
  return v[p1.second] > v[p2.second];
  // may use external v in scope
};
```

- Which lets me write

```
priority_queue<pii, vii, ComparisonClass> pq;
```

Set and Map

- I use `set<int> st;` often, but really it is just a special case of `map<int, int>`. The useful methods are `.insert(x)`, `.count(x)`, `.erase(x)`, all are $\mathcal{O}(\log n)$.
- For `map<int, int> mp;` the useful methods are `[x]`, `.count(x)`, `.erase(x)`, all are $\mathcal{O}(\log n)$.
- One subtle note: doing the check

  ```
  if (mp[x])
  ```

  automatically assigns `x` to 0 if `x` was not in `mp` before. Therefore, this check might increase the size of `mp`. The way I often do it is

  ```
  if (mp.count(x))
  ```

- To traverse through `mp` we can simply

  ```
  for (auto p : mp) cout << p.first << ' ' << p.second;
  ```

- There also exists binary search methods for maps, such as `mp.lower_bound(x)`, which is $\mathcal{O}(\log n)$.

Unordered Set and Map

- When changing the `set<int>` and `map<int, int>` to `unordered_set<int>` and `unordered_map<int, int>` all logarithmic complexities change to constant $\mathcal{O}(1)$.
- This does not necessarily mean that the code runs faster, but this is often the case.
- These structures are implemented with some associated hash function. For most basic data types, these hash functions are already implemented in C++ standard library.
- One major exception is the `pair<int, int>` type and its variants. Then you have to yourself implement a hash function (but I have never had to do this myself).

Structures

- You really should never need to create a `C++ Class`, but the `C Structs` are often useful.
- The compact syntax is

```
typedef struct Atom {
  Atom* nxt;
  int x;
} Atom;
```

- Now you can create an `Atom` by simply declaring it

```
Atom atom; atom.nxt = root; atom.x = x;
```

  or you can allocate an `Atom` on the heap and get a pointer to it

```
Atom* atom = new Atom; atom->nxt = root; atom->x = x;
```

- Remember, don't be afraid of pointer stuff. Learn them by solving Kattis problems. :)

Disjoint-Set Structures (Union-Find)

- One of the most important data structures of all time.
- The key (and bottleneck) of finding a minimal spanning tree.
- Finds the connected components of a graph.
- It consists of a vector initialized as

```
vi v(mxn); rep(i, 0, mxn) v[i] = i;
```

- It also has two methods `void union(int, int);` and `int find(int);` Naive implementation:

```
void union(int i, int j) {v[find(i)] = find(j);}
int find(int i) {while(v[i]!=i) i = v[i]; return i;}
```

  Time complexity is horrendous $\mathcal{O}(n)$ for both `union` and `find`.
- **Corpus idea:** Write an efficient union-find.

Bitsets

- I honestly have not used `bitset<mxn>` a lot, but it is *very* popular.
- It seems like they are space efficient and sometimes faster than normal bit manipulation?
- For example, you can initialize

```
bitset<8> bt(15);     bitset<16> bt("111");
```

  and then perform useful methods such as `.set(i)`, `.flip(i)`, `.count()`, `.to_string()` and `.to_ulong()`.
- I don't really have any wisdom here, but I still wanted to mention them. Instead I will take the opportunity to thank you for following the study circle and I hope you are having fun. :)

This Week

- Read Chapter 3 and 5 in Laaksonen[1].
- Read Chapter 3, 5, and 6 in Sannemo[2].
- Solve half of the weeks problems.
- If you feel inspired, begin with the corpus and upload your code to the gitlab.

[1] Antti Laaksonen. *Guide to competitive programming*. Springer, 2020.

[2] Johan Sannemo. "Principles of Algorithmic Problem Solving". In: *Draft version* (2018).

Code Review