Competitive STL Extensions

Meeting C++ 2018

Fedor Alekseev

Moscow Institute of Physics and Technology: My pity

November 16, 2018

Competitive Programming

A contest

- Participants receive a set of 4–12 problems, and have just 2–5 hours

Competitive Programming

A contest

- Participants receive a set of 4–12 problems, and have just 2–5 hours
- Problems are well-defined and usually have computer science nature
- Solving a problem means sending a program to the judging system. The program should pass all the secret test cases within time limit

A contest

- Participants receive a set of 4–12 problems, and have just 2–5 hours
- Problems are well-defined and usually have computer science nature
- Solving a problem means sending a program to the judging system. The program should pass all the secret test cases within time limit
- Optimal algorithmic complexity is usually enough, especially for C++

Competitive Programming

A contest

- Participants receive a set of 4–12 problems, and have just 2–5 hours
- Problems are well-defined and usually have computer science nature
- Solving a problem means sending a program to the judging system. The program should pass all the secret test cases within time limit
- Optimal algorithmic complexity is usually enough, especially for C++
- Solutions are compiled in a judging environment without any additional libraries, with just the stock compiler installation

Standard library

- Algorithms: `sort`, `lower_bound`, `unique`, `next_permutation`, etc
- Data structures: {`unordered_`,}{`set`,`map`}, bitset, simpler containers

Standard library

- Algorithms: `sort`, `lower_bound`, `unique`, `next_permutation`, etc
- Data structures: {`unordered_`,}{`set`,`map`}, bitset, simpler containers
- GNU C++ specific: `#include <bits/stdc++.h>` includes everything!

popcount: number of set bits

```
int main(int argc, const char* argv[]) {
  static_assert(0 == __builtin_popcount(0));  // wow so constexpr
  static_assert(4 == __builtin_popcount(0b1111));
  static_assert(3 == __builtin_popcount(0b100101));
  return __builtin_popcount(argc);
}
```

godbolts under x86 to

```
main:
        xor     eax, eax
        popcnt  eax, edi
        ret
```

Similarly, `__builtin_clz` and `__builtin_ctz` count leading/trailing zeros

SGI STL extensions: power

- Sometimes you have an operation (a^n) that can be expressed as some other operation ($a \cdot a$) repeated n times
- This is usually called exponentiation

SGI STL extensions: power

- Sometimes you have an operation (a^n) that can be expressed as some other operation ($a \cdot a$) repeated n times
- This is usually called exponentiation
- Matrix exponentiation: $A^n = E \cdot \underbrace{A \cdot A \cdot \ldots \cdot A}_{n \text{ times}}$, where E is identity matrix
- Modular exponentiation:

$$a^n \mod p = 1 \cdot \underbrace{(((a \mod p) \cdot a \mod p) \cdot \ldots \cdot a \mod p)}_{n \text{ multiplications modulo } p}$$

SGI STL extensions: power

- Sometimes you have an operation (a^n) that can be expressed as some other operation ($a \cdot a$) repeated n times
- This is usually called exponentiation
- Matrix exponentiation: $A^n = E \cdot \underbrace{A \cdot A \cdot \ldots \cdot A}_{n \text{ times}}$, where E is identity matrix
- Modular exponentiation:

$$a^n \mod p = 1 \cdot \underbrace{(((a \mod p) \cdot a \mod p) \cdot \ldots \cdot a \mod p)}_{n \text{ multiplications modulo } p}$$

- std::pow does usual multiplication of real numbers only
- If multiplication is associative, this still can be done in just $O(\log n)$ multiplications

SGI STL extensions: power

```cpp
#include <bits/extc++.h>

constexpr int64_t Modulo = 1000000007;  // a prime number
auto multiply_modulo = [](int64_t a, int64_t b) {
  return a * b % Modulo;
};
// this is required to fully define the operation
// will be called through ADL
int64_t identity_element(decltype(multiply_modulo)) {
  return 1;
}
bool fermat_little_theorem_holds(int64_t x) {  // x^p ≡ x (mod p)
  return __gnu_cxx::power(x, Modulo, multiply_modulo) == x % Modulo;
}
```

Policy-Based Data Structures

- Policy-Based Data Structures library implements several types of search trees, hash tables, and heaps in an extensible way.

Policy-Based Data Structures

- Policy-Based Data Structures library implements several types of search trees, hash tables, and heaps in an extensible way.
- Shipped with GNU C++ library as an extension within namespace `__gnu_pbds`

PBDS: order statistics tree

- Usual std::set maintains a dynamic sorted sequence. It has methods like

  `iterator set<T>::lower_bound(const T&) const`

  but no methods like

  `iterator set<T>::at(size_t) const`

PBDS: order statistics tree

- Usual std::set maintains a dynamic sorted sequence. It has methods like

  ```
  iterator set<T>::lower_bound(const T&) const
  ```

  but no methods like

  ```
  iterator set<T>::at(size_t) const
  ```

- Efficient ($O(\log n)$) implementation of these methods would require maintaining additional information in the search tree nodes

PBDS: order statistics tree

- Usual std::set maintains a dynamic sorted sequence. It has methods like
  `iterator set<T>::lower_bound(const T&) const`
  but no methods like
  `iterator set<T>::at(size_t) const`
- Efficient ($O(\log n)$) implementation of these methods would require maintaining additional information in the search tree nodes
- `__gnu_pbds::tree_order_statistics_node_update` is a tree update policy that does exactly that, and enables methods
  `tree::iterator tree::find_by_order(size_t) const`
  and
  `size_t tree::order_of_key(const T&) const`

PBDS: order statistics tree declaration

```cpp
#include <bits/extc++.h>
using namespace __gnu_pbds;

template<typename K, typename V, class Earlier = std::less<K>>
using OrderStatsMap = tree<
  K, V, Earlier,
  rb_tree_tag,  // or splay_tree_tag
  tree_order_statistics_node_update  // extension policy
>;

template<typename K, class Earlier = std::less<K>>
using OrderStatsSet = OrderStatsMap<K, null_type, Earlier>;
```

PBDS: order statistics tree usage

```
OrderStatsSet<int> s;
for (auto k : {12, 505, 30, 100}) {
  s.insert(k);
}

// The order of the keys should be: 12, 30, 100, 505.
assert(12 == *s.find_by_order(0));
assert(100 == *s.find_by_order(2));
assert(s.end() == s.find_by_order(4));

assert(0 == s.order_of_key(10));
assert(1 == s.order_of_key(30));
assert(4 == s.order_of_key(1000));
```

Lacking utilities

- C++ is a great language for competitive programming, but

Lacking utilities

- C++ is a great language for competitive programming, but
- There are some lacking utilities that still constrain its dominance

Lacking utilities

- C++ is a great language for competitive programming, but
- There are some lacking utilities that still constrain its dominance
- Most notably, arbitrary precision arithmetics: sometimes it is pragmatic to switch to Python or Java just for big integers

kthxbye

- Thanks!
- More examples:
  `https://github.com/moskupols/competitive-stl-extensions`
- For more info on PBDS see GNU C++ library manual:
  `https://goo.gl/PmR86Z`