Competitive Programming Workshop

Day 2

by Giacomo Fabris, Francesco Lotito (University of Trento)
on 2020-07-15

Previous week's contest

Basic concepts

STL wonderland

Union-Find Disjoint Set

Previous week's contest

» Problem C - hello

» Problem F - inflation

Greedy matching between canisters and balloons: *assign the smallest canister to the smallest balloon.*

1. Sort input vector;
2. loop from 1 to N, compute the minimum inflation ratio, quick fail if the canister is too big.

» Problem E - kleptography

We receive as input a ciphertext b of length m and the last n characters of the plaintext a. We shall decipher the whole plaintext. The key k is composed of an initialization vector of length n, then, $k[n+i] = a[i]$. The encryption function is: $b[i] = (a[i] + k[i]) (\text{mod } 26)$

We see that:

$$
\begin{aligned}
k[i] &= b[i] - a[i] (\text{mod } 26) \\
a[i] &= k[i+n] \\
\Rightarrow a[i] &= (b[i+n] - a[i+n]) (\text{mod } 26)
\end{aligned}
$$

» Problem D - jabuke

Area calculation is straightforward - just compute it using the provided formula.
To test whether a point $P := (x, y)$ is inside or outside the triangle ABC with vertices $A := (x_A, y_A), \ldots$, you have (at least) two options:

1. sum areas of triangles ABP, ACP, BCP; see if it matches area of triangle ABC
2. compute the implicit function of rays passing through AB, BC, AC; P is inside the triangle if $r_{AB}(x_P, y_P) \cdot r_{AB}(x_C, y_C) \geq 0$, and the same shall hold for r_{BC} w.r.t. vertex A and r_{AC} w.r.t. vertex B.

» Problem B - divisors

#1

Definition of combinations:

$$\binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Always remember this handy formula:

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

Combinations grow very fast, you cannot compute them directly.

» Problem B - divisors

#2

The easy way to do this:

1. factorize each term separately; all numbers are ≤ 431 — precompute all primes up to 431 using e.g. Sieve of Erathostenes;
2. memorize the exponent *n* of each prime number (sum all exponents of the numerator, subtract those of the denominator);
3. every prime number could not be chosen, or chosen up to *n* times.

» Problem B - divisors #2

The easy way to do this:

1. factorize each term separately; all numbers are ≤ 431 — precompute all primes up to 431 using e.g. Sieve of Erathostenes;
2. memorize the exponent n of each prime number (sum all exponents of the numerator, subtract those of the denominator);
3. every prime number could not be chosen, or chosen up to n times.

Some optimization tricks may be necessary to make this solution pass the time limit, e.g.:

* load a static array / constexpr containing the prime numbers, do not execute the sieve at runtime;
* note that the first k terms of the factorial are always simplified and therefore shall not be factorized;
* w.r.t. the previous suggestion, remember that $\binom{n}{k} = \binom{n}{n-k}$

» Problem B - divisors

#3

The clever way to do this: open Wikipedia and search for **Legendre's formula**.
It will speed-up calculation for this problem.

Basic concepts

» Big O notation

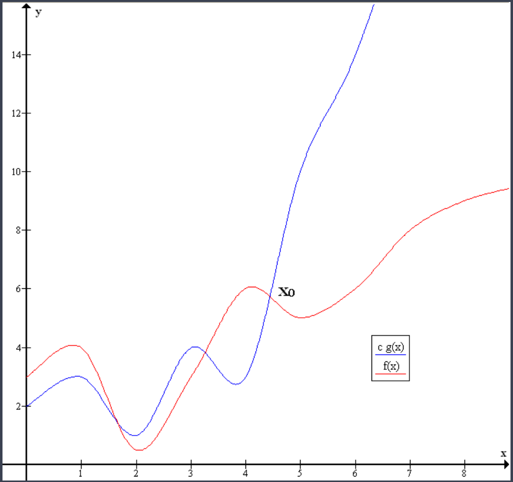


Figure: $f(x) \in \mathcal{O}(g(x))$ as there exists $c > 0$ and x_0 such that $f(x) \leq cg(x) \quad \forall x \geq x_0$.

» Aggregate analysis

We may obtain tighter upper bounds if we know the frequency of worst-case executions.
Aggregate analysis evaluates the upper bound averaging on multiple executions.

STL wonderland

» std::vector

Rationale

Elements are indexed and stored contiguously. Size of underlying array is automatically handled.

» std::vector

Rationale

Elements are indexed and stored contiguously. Size of underlying array is automatically handled.

Interface

* [ ] operator - access element at index (note: undefined behaviour if index > underlying array size) - $\mathcal{O}(1)$
* $push\_back(elem)$ - append element to the end - amortized $\mathcal{O}(1)$
* $assign(n, elem)$ - initialize vector of size n assigning each cell the element $elem$. Second parameter is optional - $\mathcal{O}(n)$ Note: same arguments of constructor.

» std::vector usage

Sort vector

```
std::sort(v.begin(), v.end());
```

Sort vector of custom type

```
typedef pair<int, int> ii;
bool mycomp(const ii a, const ii b) {
    return a.first < b.first;
}
//...
std::sort(v.begin(), v.end(), mycomp);
```

» std::vector usage

Sort vector of custom type, lambda flavour

```
std::sort(v.begin, v.end(), [](const ii a, const ii b) {
    return a.first < b.first;
});
```

Initialize DP matrix

```
v.assign(N, vector<int>(N, -1));
```

» std::queue, std::stack

Rationale

Queues *and* stacks *are* really nothing more than *a* vector with *a* (slightly) different interface.

» std::priority_queue

Rationale

A queue in which elements are sorted. If two elements have the same priority, then FIFO.
Implemented (by default) *as a* std::vector *and a* heap.

» std::priority_queue

Rationale

A queue in which elements are sorted. If two elements have the same priority, then FIFO.
Implemented (by default) as a std::vector and a heap.

Interface

* *empty*(): test empty - $\mathcal{O}(1)$
* *push*(*elem*): insert element - amortized $\mathcal{O}(\log n)$
  ($\mathcal{O}(\log n)$ heap insertion + amortized $\mathcal{O}(1)$ vector push)
* *front*(): access top element - $\mathcal{O}(1)$
* *pop*(): remove top element - $\mathcal{O}(1)$

» std::priority_queue usage

Dijkstra's prioq

Use *std* :: *greater*, which is the default, reverse-order comparator for built-in types (works also with *std* :: *pair*!)

```
typedef pair<int, int> ii;
//...
priority_queue<ii, std::vector<ii>, std::greater<ii>> pq;
pq.push(ii(0, 0));
```

» std::set, std::unordered_set

Rationale

Containers that store unique values, and which allow for fast retrieval of individual elements based on their value.

* std::set are ordered (trees!)
* std::unordered_set are unordered (hash maps!)
* std::multiset and std::unordered_multiset may have non-unique values

» std::set, std::unordered_set

Rationale

Containers that store unique values, and which allow for fast retrieval of individual elements based on their value.

* std::set are ordered (trees!)
* std::unordered_set are unordered (hash maps!)
* std::multiset and std::unordered_multiset may have non-unique values

Interface

* *find*(*elem*): find element - returns an iterator (set::end if not found)
* *insert*(*elem*): inserts element (if exists and set is not multi, no change)

» std::set, std::unordered_set usage

Notes on complexity

* Average case, insertion in a set is $\mathcal{O}(log\ n)$, accessing an element is $\mathcal{O}(log\ n)$.
* Average case, insertion in an unordered_set is $\mathcal{O}(1)$, accessing an element is $\mathcal{O}(1)$
* $\rightarrow$ in competitive programming always use unordered_set if order does not matter and you do not need to access elements sequencially

» Set operations

Set intersection

```
unordered_set<int> sa, sb, si;
set_intersection(sa.begin(),sa.end(),sb.begin(),sb.end(),
    std::inserter(si,si.begin()));
```

Similarly, there is also
And also: *set_difference*, *set_simmetric_difference*, *set_union*.
Complexity: linear in the cost of insertion and access to the set ($\mathcal{O}(N)$ if unordered, $\mathcal{O}(N \log N)$ if ordered)

» std::map, std::unordered_map

Rationale

* A set for the keys and an ancillary data structure storing the valueassociated to each key.
* We have std::map, std::unordered_map, std::multimap, std::unordered_multimap

Interface

Not that different from a set, note:

* In a $map < K, V >$ you shall insert $std :: pair < K, V >$. Using a typedef is probably the fastest way to do it.
* You can access directly a value using the $[]$ operator, passing the desired key between brackets. While this may seem fancy, note that it has a strange behaviour: if you access a key which is not in the map, a new element is inserted, regardless whether an r-value has been passed! (In that case, constructor default will be used)
* $Rightarrow$ test if a key is in the map using $map.find()! = map.end()$ (it returns the iterator of the element)

Union-Find Disjoint Set

» Intro

STL is nice, its data structures are general-purpose and have a wide range of applications.
However, in CP, sometimes (often) it is necessary to use more tailored data structures.
Today we will see one of the many well known DS, not implemented in the STL: the UFDS.

» Definition

A Union-Find Disjoint Set is *a* d*a*t*a* structure which models *a* **collection of disjoint sets**.
As the name suggests, the following operations are usually implemented:

* initialize the UFDS with n sets $\{1\}, \{2\}, \ldots, \{n\}$ in $\mathcal{O}(n)$

» Definition

A Union-Find Disjoint Set is *a* d*a*t*a* structure which models *a* **collection of disjoint sets**.
As the n*a*me suggests, the following operations *a*re usually implemented:

* initi*a*lize the UFDS with n sets $\{1\}, \{2\}, \ldots, \{n\}$ in $\mathcal{O}(n)$
* **union**: merge two sets of the UFDS in (*a*lmost) $\mathcal{O}(1)$

» Definition

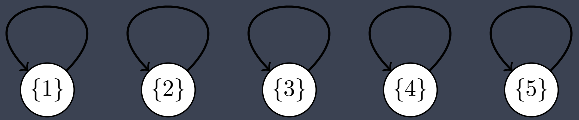
A Union-Find Disjoint Set is a data structure which models a **collection of disjoint sets**.
As the name suggests, the following operations are usually implemented:

* initialize the UFDS with n sets $\{1\}, \{2\}, \ldots, \{n\}$ in $\mathcal{O}(n)$
* **union**: merge two sets of the UFDS in (almost) $\mathcal{O}(1)$
* **find**: determine which set an item belongs to / determine if two items belong to the same set in $\mathcal{O}(1)$

Using the STL (e.g. vector of unordered sets?) these operations would be slower!

» Idea

The UFDS data structure is stored as a forest, i.e. every disjoint set is a tree, where the root, the "representative" element, is just one element of the set.
The forest is memorized as a vector V of n integers. $V[i]$ contains the parent of the tree of the element i; if i is a root then $V[i] = i$.
When the UFDS is initialized, $V[i] = i \ \forall i$.



» Operations

`Root(x)`: return the representative node of the set in which x is. Simple tree visit.

» Operations

`Root(x)`: return the representative node of the set in which x is. Simple tree visit.

`Union(x, y)`: merge the two trees which contain the element x and y, which means, change the representative of y to be x (or the other way around). Heuristic improvement: perform the merge which minimizes the resulting depth.

» Operations

`Root(x)`: return the representative node of the set in which *x* is. Simple tree visit.

`Union(x, y)`: merge the two trees which contain the element *x* and *y*, which means, change the representative of *y* to be *x* (or the other way around). Heuristic improvement: perform the merge which minimizes the resulting depth.

`Find(x, y)`: `Root(x) == Root(y)`

» Path compression

Whenever we find the representative (root) item of a disjoint set by traversing the tree from the leaves to the root, we can set the parent of all items traversed to point directly to the root.
All subsequent find operations may be performed with single access to the vector V.

» Example

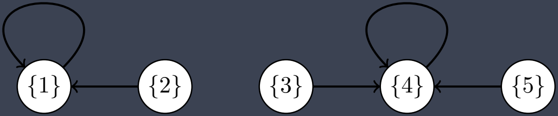


Figure: Sets: $\{\{1,2\},\{3,4,5\}\}$

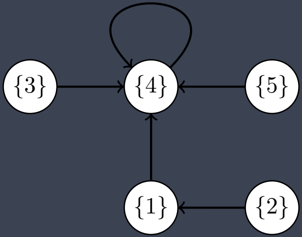


Figure: `Union (2, 4)`

» Example

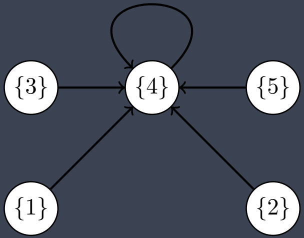


Figure: Path compression after `Find (2, 4)`

Thanks!

Today’s contest:
`https://open.kattis.com/contests/ucen9t`