On data structures

STL, Prefix sums, Fenwick tree

Giulia Carocari, Giacomo Fabris, Francesco Lotito

Università degli Studi di Trento

ICPC Training @ UniTN
Day 4 - April 04, 2019

Table of Contents

Table of Contents

A - Union-Find

Problem definition

Given a Union-Find data structure, perform merges ($=ab$ means merge the sets containing a and b) and find operations. For every query $?ab$, output whether a and b belong to the same set.

A - Union-Find

Problem definition

Given a Union-Find data structure, perform merges ($=ab$ means merge the sets containing a and b) and find operations. For every query $?ab$, output whether a and b belong to the same set.

Solution

Implement the Union-Find Set data structure and perform the queries. Don't forget to implement *path compression* and *rank heuristics* as well!

Programming tip: The problem may ask you to perform a huge amount of queries per dataset: in this case I/O becomes the bottleneck of the execution. Use the faster `scanf`/`printf` functions rather than the standard `C++` `cin`/`cout` operators.

B - Minimum Spanning Tree

Problem definition

You are given a weighted undirected graph and are requested to print the weight of the MST and its edges, sorted in lexicographically increasing order of their ends.

B - Minimum Spanning Tree

Problem definition

You are given a weighted undirected graph and are requested to print the weight of the MST and its edges, sorted in lexicographically increasing order of their ends.

Solution

Run Kruskal/Prim's Algorithm on the input graph, sort and output the resulting tree. Couldn't be any simpler.

C - Arctic Network

See last week's slides for this one.

D - Freckles

Problem definition

You are given a series of coordinates for freckles on a person's back. You want to know the minimum amount of ink you have to use in order to connect them all (i.e. to draw a tree with the freckles as vertices).

D - Freckles

Problem definition

You are given a series of coordinates for freckles on a person's back. You want to know the minimum amount of ink you have to use in order to connect them all (i.e. to draw a tree with the freckles as vertices).

Solution

This is a minimum spanning tree problem for a complete graph on the Carthesian plane (just like Arctic Network). Build the graph by connecting each dot to every other freckle and then output the value returned by Prim's/Kruskal's Algorithm.

E - Driving Range

Problem definition

Cities in a country are connected by roads of different legths. In every city there is a loading station for electric cars. You want to know the minimum driving range of such a car in order to allow its owner to travel from a city to any other. If necessary the driver may stop in a city to charge the battery.

E - Driving Range

Problem definition

Cities in a country are connected by roads of different legths. In every city there is a loading station for electric cars. You want to know the minimum driving range of such a car in order to allow its owner to travel from a city to any other. If necessary the driver may stop in a city to charge the battery.

Solution

Just another MST problem. This time you need to print the weight of the heaviest/longest edge in the minimum spanning tree.

F - Single source shortest path, non-negative weights

Problem definition

Given a graph and one of its vertices marked as *source*, you want to know the length of the shortest path from the source to some other nodes. If there is no path to a node, then print `Impossible`.

F - Single source shortest path, non-negative weights

Problem definition

Given a graph and one of its vertices marked as *source*, you want to know the length of the shortest path from the source to some other nodes. If there is no path to a node, then print `Impossible`.

Solution

Run Dijkstra's algorithm on the graph, thus filling a distance vector d. For each node u that is queried simply print $d[u]$, or `Impossible` if $d[u] = +\infty$.

G - Flowery Trails

Problem definition

People are lazy, and when they visit the park they only want to reach the highest peak following one of the shortest paths that connects the entrance to the POI. You want to cover these shortest paths with flowers (on both sides), so you need to know how many metres of greens you need to buy in order to do so.

G - Flowery Trails

Problem definition

People are lazy, and when they visit the park they only want to reach the highest peak following one of the shortest paths that connects the entrance to the POI. You want to cover these shortest paths with flowers (on both sides), so you need to know how many metres of greens you need to buy in order to do so.

Solution

We need to find all eges of the *park-graph* that belong to a shortest path from the entrance to the highest peak. The trick is to run Dijkstra's Algorithm twice, once using the entrance as a source (filling vector d) and once using the highest peak (filling d').
Once we have done this, an edge (u, v) belongs to a sorthest path if

$$d[u] + w(u, v) + d'[v] = d[peak] \vee d[v] + w(u, v) + d'[u] = d[peak]$$

H - Brexit

Problem definition

We are given a set of countries (nodes), connected by trading partnerships (edges) of countries in a Union. One day, one country (*cough-cough*) decides to leave the Union, and all other countries, worried about economics, might decide to leave as well. A country will leave the Union if at least half its trading partners are leaving. Will your home country be still in the Union after the chain reaction created by the first State?

H - Brexit

Solution

Keep two vectors, one to count the initial number of trading parters for each country and the other to know how many parters it has left.
Use a BFS to simulate the process of the leaving country starting from the UK. If the number of remaining partners for the first country in the queue is less or equal than half the size of the original set, remove this node and update it's neighbors' counters. In the end check if

$$remaining[homeCountry] \leq parters[homeCountry]$$

et voilà.

I - Brexit Negotiations

Problem definition

We need to make Brexit Negotiations as short as possible, by selecting an order to discuss topics.

- Some topics need to be discussed before other (but there are no cycles in these dependencies).
- Each discussion lasts a certain amount of time and is preceeded by a 1-minute recap for each topic that was examined earlier.

How *short* can the longest of all meetings be in the optimal case?

I - Brexit Negotiations

Solution

Ideally, we want to find a topological ordering of topics such that

$$max(w(topsort(i)) + i)$$

is minimal.
We can build such an ordering in a bottom-up fashon: start from all those discussions that do not need to preceed any other topic; the first one we consider is the last one that will actually be scheduled, meaning that it will be preceeded by a n minute recap.
At this point we make a greedy choice: we sort the topics that are available by increasing length (`priority_queue`). For every topic u we schedule, we decrease the degree of its neighbors v, and once $deg(v) = 0$ we insert v into the queue.

I - Family DAG

Problem definition

The input is indeed a family graph of parent-child relationships. Your task is to find all the people in the graph that are either their own ancestors (**paradox**) or that have the same person more than once among their ancestors (**hillbilly**).

I - Family DAG

Problem definition

The input is indeed a family graph of parent-child relationships. Your task is to find all the people in the graph that are either their own ancestors (**paradox**) or that have the same person more than once among their ancestors (**hillbilly**).

Solution

Build the graph so that for every parent-child relationship there is a child-parent edge in the graph. Then, for every person ($N \leq 100$), visit the graph. If you visit the source more than once you have encountered a paradox. If you happen to visit any other node more than once, then the source of the visit is a hillbilly.

Note: *paradox* $\Rightarrow$ *hillbilly*, therefore, if you use two boolean variables for the two properties check for hillbilly only if the person is not a paradox.

Table of Contents

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.

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

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 intersection

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

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, $\mathcal{O}(NlogN)$ 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 a r-value has been passed! (In that case, constructor default will be used)

Table of Contents

Towards "competitive" data structures

- What we have seen until now is very standard.
- Some problems may require (more or less) advanced data structures.
- For example, we may be interested in extracting the minimum / the sum / ... of a list of elements (Yes, of course it could be done in $\mathcal{O}(N)$, but we can do *way* better).

All information which could be extracted from a list exploiting a divide-et-impera technique is a candidate for the usage of more advanced techniques.

Prefix sum

Problem

Given a list of n integers $a[0], \dots, a[n-1]$ and k queries, consisting of a pair (l_k, r_k); $l_k \leq r_k$, find for all k the value $s_k = a[l_k] + \dots + a[r_k - 1]$.

Approach #0

- Scan the list: $\mathcal{O}(n \cdot k)$.

Prefix sum

A better approach

- Compute in $\mathcal{O}(n)$ an auxiliary list P, such that $P[i] = a[0] + \cdots + a[i-1] = P[i-1] + a[i-1]$
- Solve all queries in $\mathcal{O}(k)$: $s_k = P[r_k] - P[s_k]$

Problem

What if "queries" operations are interleaved with "update" operations?

Generalization

It can be shown that this (and the following) techniques may be extended beyond sums - for every associative and invertible operator.

Table of Contents

Fenwick tree

Problem

We are given an array P of n integers and

- k queries of the form: "given l and r, $l < r$, compute $P[l] + \cdots + P[r-1]$"
- h queries of the form: "given x and v, assign $P[x] = v$".

Queries of the two types may be interleaved in any order.

Introduction

- A Fenwick Tree is an array of numbers, indexed from 1 to n.
- We need to find a smart way to compute prefixes, so that both queries may be resolved in $\mathcal{O}(log\ n)$.
- We will see that every element of the original array is covered by approx. $log\ n$ elements of the original array, yielding the desired result.

Construction

The Fenwick Tree associated to an array *a* of length *n* is an array *ft* of length *n* (both indexed from 1 to *n*) such that:

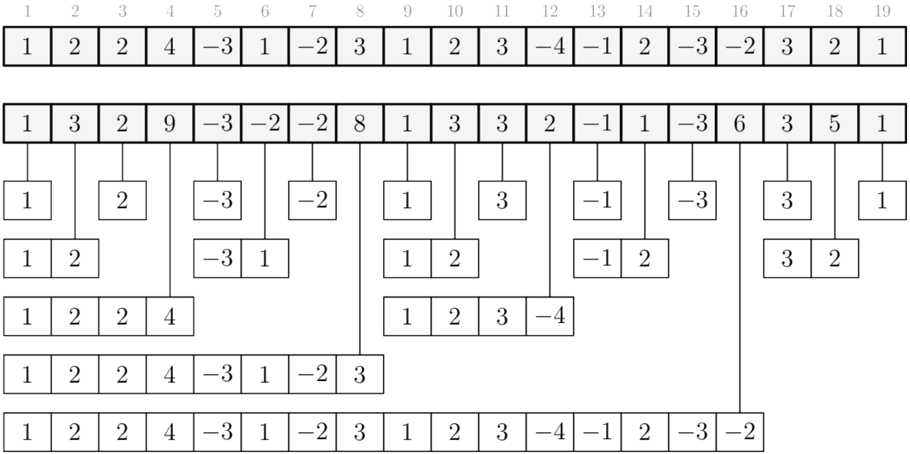
$$ft[i] = \sum_{\bar{i}+1}^{i} a[i]$$

where $\bar{i} = i - lsb(i)$

Programming tip

```
lsb(i) = i & (-i)
```

Fenwick trees



Fenwick tree · prefix sum

Compute prefix sum

The prefix sum $a[1] + \cdots + a[i]$ can be calculated as follows:

$$prefix(i) = \begin{cases} 0 & \text{if } i = 0 \\ ft[i] + prefix(i - lsb(i)) & \text{otherwise} \end{cases}$$

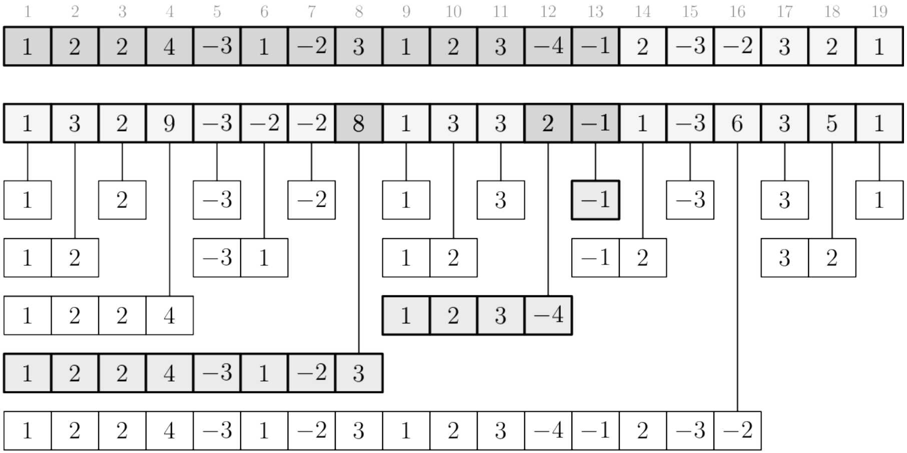
It can easily be shown that the number of terms in this summation is equal to the number of ones of i, which means, $\mathcal{O}(log\ n)$.

Example

See next slide: calculate

$$prefix(13) = ft[13] + prefix(12) = \cdots = ft[13] + ft[12] + ft[8]$$

Fenwick tree · prefix sum



Fenwick tree · range query

```
#define lsb(x) (x & (-x))

int prefix_sum(size_t k) {
        int ans = 0;
        for (; k != 0; k -= lsb(k)) {
                ans += ft[k];
        }
        return ans;
}

int sum(size_t a, size_t b) {
        return prefix_sum(b) - prefix_sum(a - 1);
}
```

Fenwick tree · point update

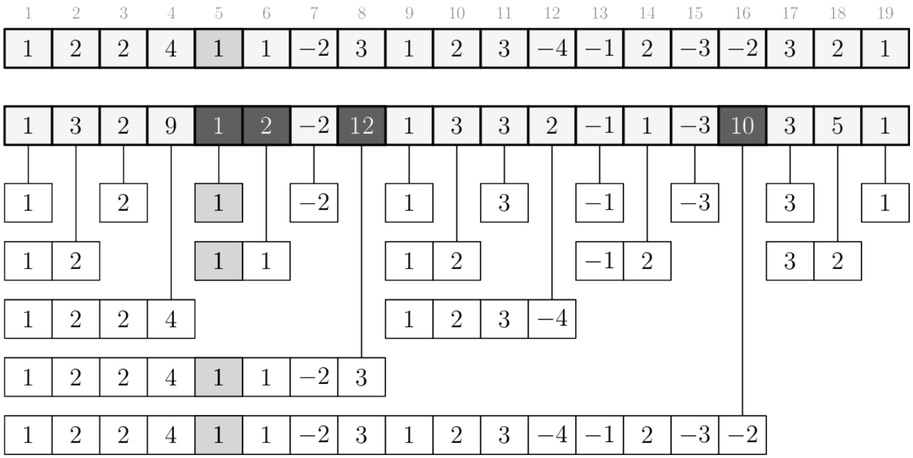
Compute point update

We want to increase the element i by a value δ. To update an element of the original array we need to update $O(log\ n)$ elements of the Fenwick tree, as follows:

$$\begin{aligned} i_1 &= i & ft[i_1] &+= \delta \\ i_2 &= i_1 + lsb(i_1) & ft[i_2] &+= \delta \\ & \dots \end{aligned}$$

until $i_k <= n$

Fenwick tree · point update



Fenwick tree · point update

```
void update(size_t k, int delta) {
        for (; k <= n; k += lsb(k)) {
                ft[k] += delta;
        }
}
```

Fenwick tree

Final remarks

- Initialize a Fenwick tree in $\mathcal{O}(n \cdot log\ n)$ by repeatedly calling the update function.
- Fenwick tree could be made zero-based tweaking the implementation of the interface.
- A way smarter trick is to use a Fenwick tree to support **point query and range update**.

Fenwick tree for range update

Problem

We are given an array P of n integers and

- k queries of the form: "given an integer δ; l and r, $l < r$, assign $P[l]+=\delta; \ldots; P[r-1]+=\delta$"
- h queries of the form: "given x, retrieve $P[x]$".

Queries of the two types may be interleaved in any order.

Fenwick tree for range update

Idea

- Given an array a of length n, start by creating the array of the finite difference d, s.t.

$$d[i] = \begin{cases} a[i] & \text{if } i = 1 \\ a[i] - a[i-1] & \text{otherwise} \end{cases}$$

- In the array a, point query can be done as follows: $a[i] = d[1] + \cdots + d[i]$
- In the array a, range update (add δ between l and r) can be done as follows: $d[l] += \delta$, $d[r+1] -= \delta$.

Fenwick tree for range update

Build

Simply, create a fenwick tree on the finite difference array d. Note that:

- a point query on a corresponds to a range query (range sum) on d
- a range update (range sum) on a corresponds to a point update on d.

Fenwick tree for range update

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	2	1	1	1	1	−2	2	2	2	2	2	2	3	3	4	4	1
1	1	0	−1	0	0	0	−3	4	0	0	0	0	0	1	0	1	0	−3

Figure: Array *a* (above), finite differences array *d* (below)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	2	2	1	1	2	2	−1	3	3	3	3	2	2	3	3	4	4	1
1	1	0	−1	0	1	0	−3	4	0	0	0	−1	0	1	0	1	0	−3

Figure: Range update: $\delta = 1, l = 6, r = 12$