Competitive Programming Algorithms

David del Val

August 26, 2021

Contents

Chapter 1

Range queries

1.1 Types of range queries

Depending on the type of function whose value we have to calculate over the given range:

- *Idempotent functions.* These functions fulfill the following condition $f(f(x)) = f(x)$. For instance, the maximum and minimum of a list of values are idempotent. Furthermore, since the GCD and LCM can be seen as the "maximum" and the "minimum" of the exponents in the prime factorization, it is only natural that they fulfill the same property.

  The most important property of these functions for range queries is that we can evaluate the elements of the range several times without affecting the result. For instance:

  $$\min(a_l, \ldots, a_r) = \min\left(\min(a_l, \ldots, a_k), \min(a_k, \ldots, a_r)\right) \quad l < k < r$$

  even though the union of the intervals includes a_k twice. This is the property in which sparse tables are based.

- *Reversible functions.* Functions that have an inverse also have advantages in range queries. That is because we can easily remove elements from any range to obtain a smaller one. In these cases we can obtain the final range by adding and subtracting ranges that have been precalculated.

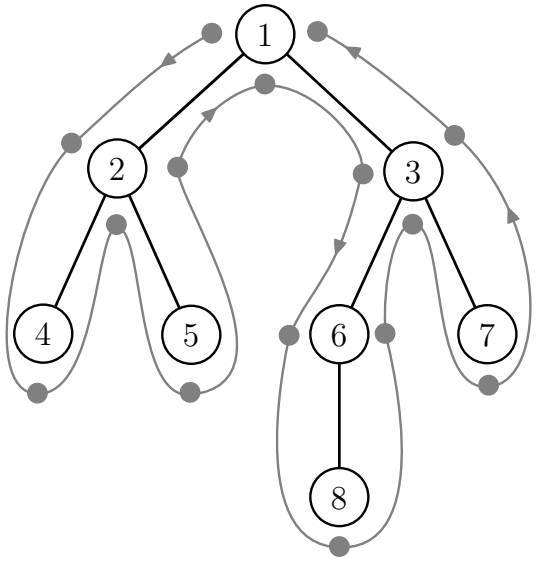
  For instance, the addition, subtraction and xor operations have an inverse (subtraction, addition and xor respectively).

Many problems can be transformed into a range query easily.

1.1.1 LCA using RMQ

To use this algorithm we need a rooted tree with undirected edges where the nodes are labeled in a particular order. Specifically, nodes at a greater depth have to be labeled with a greater value. Furthermore, the algorithm could be extended to process a forest (instead of a single tree) but this version cannot process forests.

We can traverse the tree using dfs and store in an array each node that we encounter. If we also keep track of the time at which each node was found for the first time, the LCA can be converted into calculating the minimum in an array of length $E + V$.



After running a DFS in the tree, we would get the following array of values and discovery times:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	4	2	5	2	1	3	6	8	6	3	7	3	1

Node	Time	Node	Time
1	0	5	4
2	1	6	8
3	7	7	12
4	2	8	9

Now if we want to get the LCA between nodes 5 and 6, we have to obtain the minimum value in the range [4, 8]. As we can see, the result is obviously 1.

If we compare this approach to binary lifting, we observe that we have a considerably more expensive precalculation phase but, afterwards, answering any queries takes constant time instead of logarithmic time. In particular, the complexities for this method are:

- Precomputate the values: $\mathcal{O}(E \log(E))$
- Calculate the LCA of a node: $\mathcal{O}(1)$
- Memory usage: $\mathcal{O}(E \log(E))$

Where $E =$ edges.

1.2 Sparse table

The sparse table is a data structure that calculates the value of a function over a range of immutable elements with the following complexity:

- $\mathcal{O}(n \log n)$ to build the data structure
- $\mathcal{O}(1)$ to answer queries on a range.
- $\mathcal{O}(n \log n)$ memory

Furthermore, the function must be **idempotent**. This allows the use of overlapped intervals in the calculations. A Sparse Table can be adapted to answer queries with more general functions. However, it would lose the constant time query capabilities and would be no more efficient than a segment tree or a BIT.

On the other hand, the data structure is rather simple, and its implementation is considerable shorter than any other alternatives.

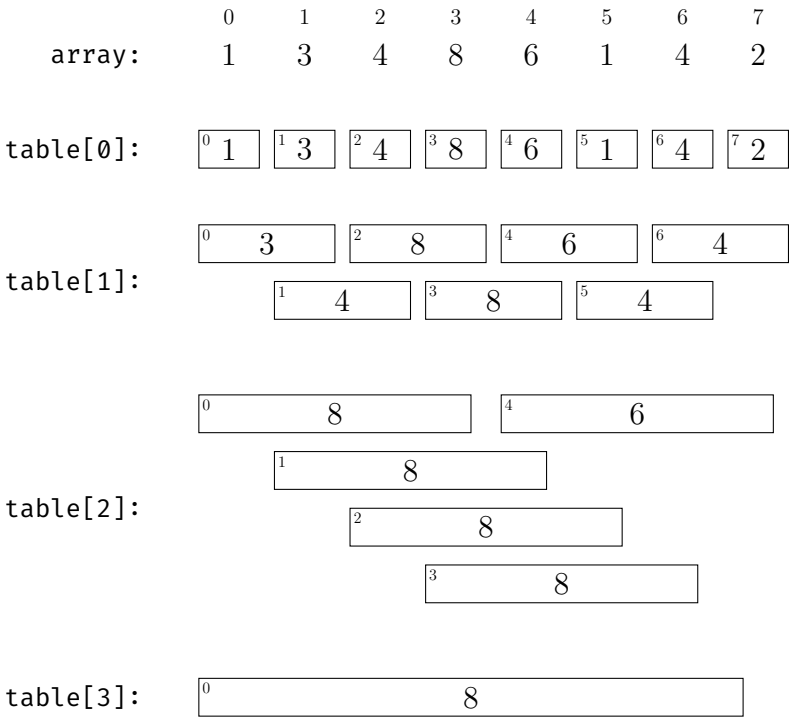
The table consists of $\log n$ rows. Each one of them contains the value of the function in intervals of length 2^j where j is the row index. In particular, $\text{table}[j][i] = f([i, i+2^j])$, where $f(I)$ is the value of f over all the elements in I.

To construct the table we take advantage of the fact that the intervals of each row are the exact union of two intervals of the previous row. Finally, during the look-up process, to obtain the logarithm of each value, we can calculate the most significant bit that is set using `__builtin_clz()`.

1.2.1 Example

One of the best applications of the sparse table is to calculate the maximum of any range in an array of elements. The following illustration displays all the intervals that will be included in the sparse table. To avoid intersections, intervals that belong to the same level of the table may be displayed above or below each other.

In each interval, we can see both the index (top left corner) and the value of the maximum in that interval. It can also be seen that all intervals can be calculated using the values obtained in the previous level.



Now, if we want to calculate the maximum between indices 2 and 7 we will:

1. Calculate the length of the interval. In this case, $7 - 2 + 1 = 6$
2. Calculate the maximum value of k such that $2^k < 6$. In this case, $k = 2$.
3. Pick the interval that starts on 2 with length $2^k = 4$ and the interval that starts on $7 - 2^k + 1$ with the same length. That is to say, intervals 2 and 4 from level 2 of the table.

   As we can see these intervals cover the entire range, albeit they overlap. However, the overlap does not affect the result of the maximum.
4. Take the maximum between the values of the two intervals selected: $\max(8, 6) = 8$

```cpp
template <typename T> class SparseTable {
private:
    vector<vector<T>> table;
    std::function<T(T, T)> f;

public:
    SparseTable(vector<T> &data, std::function<T(T, T)> f) : f(f) {
        int n = data.size();
        table.pb(data);
        for (int j = 1; (1ll << j) <= n; ++j) {
            vector<T> nextRow(n);
            for (int i = 0; i + (1ll << j) <= n; ++i) {
                int otherIndex = i + (1ll << (j - 1));
                nextRow[i] = f(table.back()[i], table.back()[otherIndex]);
            }
            table.push_back(std::move(nextRow));
        }
    }
    static int msb_index(int x) { return __builtin_clz(1) - __builtin_clz(x); }

    T valueInRange(int left, int right) {
        int j = msb_index(right - left + 1);
        ll intervalSize = 1ll << j;
        return f(table[j][left], table[j][right - intervalSize + 1]);
    }
};
```

1.2.2 Sparse Table (sort of) with $\mathcal{O}(n)$ memory

This variant of the sparse table is only capable of answering very specific queries. In particular, the operation must meet the following constraints:

- It is idempotent (as in a normal Sparse Table)
- The result of the operation over a range is an element in that range

Therefore, this data structure will mostly only be used for querying the maximum or the minimum over a range. On the other hand, we obtain a data structure that uses significantly less memory and can be built in considerably less time, albeit with slower queries.

Explanation

With that goal in mind, we will group the elements of the array in blocks. These blocks should be of size $\log(n)$. However, since n will never be larger than 2^{30}, we can set their size to 30. Now we will build two data structures:

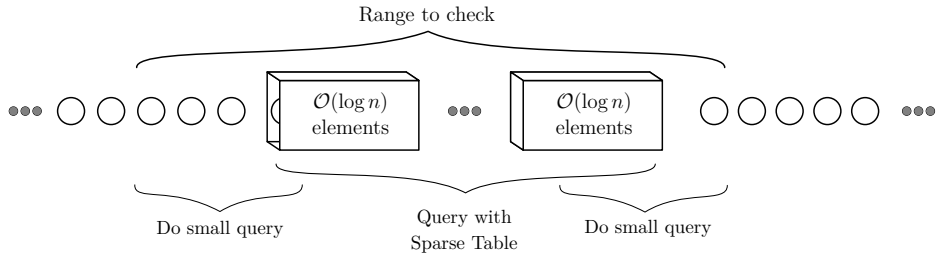
- A Sparse Table where each element is a block of the array. In order to simplify this task, in the Sparse Table we will store the index of the result of applying the operation to the range instead of the result itself.

  Therefore, the elements in the first row of the table will be the index of the result of applying the operation to each block of the array while the elements in the following layers will represent applying the operation to 2, 4, 8 …blocks.
- A structure that stores the result for every range of 30 contiguous elements in the array.

If we have these two data structures we can split the (large) query into three subqueries that will most likely overlap. In particular, we will split the range $[l, r]$ into:

- A range of size 30 starting at l.
- The blocks that are fully contained within the interval.
- A range of size 30 that ends at r.

The middle range will be processed using the Sparse Table of blocks while the other two must be calculated using the new data structure.



In order to answer the small queries, we will use a bitmask for every position. That is to say, a vector of n integers where each of them is a bitmask for that position. Let's look at the bitmask of position r. The bit $r - k$, $(k < 30)$ will be one iff that element is the result in the range $[r - k, r]$. For instance, at position 6 of this example array the bitmask would be: This bitmask approach is particularly useful because the vector of bitmasks can

	0	1	2	3	4	5	6
array:	1	4	2	6	7	4	5
mask:	1	0	1	0	0	1	1

be built in linear time. If we look at the code below (`build_mask()`), the `while` loop will only be executed at most once for every element while the `for` loop iterates through all elements exactly once.

```
20  class rmq {
21      vl data;
22      int n, rowSize;
23      vi mask, sparseTable;
24      static const int blockSize = 30;
25      std::function<ll(ll, ll)> f;
26
27  public:
28      rmq(const vl &v, decltype(f) function)
29          : data(v), n(v.size()), mask(n), sparseTable(n), f(function) {
30          rowSize = n / blockSize;
31          build_mask();
32          build_sparse_table();
33      }
34
35      ll query(int l, int r) {
36          // Small query
37          if (r - l + 1 <= blockSize) {
38              return data[small_query_index(r, r - l + 1)];
39          }
40
41          // Get the result for the endpoints
42          int ans =
43              op(small_query_index(l + blockSize - 1), small_query_index(r));
44
45          // Query the sparse table
46          int x = l / blockSize + 1, y = r / blockSize - 1;
47
48          if (x <= y) {
49              // Row to query
50              int j = msb_index(y - x + 1);
51
```

```
            ans = op(ans, op(sparseTable[rowSize * j + x],
                             sparseTable[rowSize * j + y - (1 << j) + 1]));
        }
        return data[ans];
    }

private:
    static int lsb(int x) { return x & -x; }
    static int msb_index(int x) { return __builtin_clz(1) - __builtin_clz(x); }

    // Get the index that contains the answer for the small query
    int small_query_index(int r, int size = blockSize) {
        int dist_from_r = msb_index(mask[r] & ((1 << size) - 1));

        return r - dist_from_r;
    }

    // Return the index that contins the result of the operation
    int op(int index1, int index2) {
        return f(data[index1], data[index2]) == data[index1] ? index1 : index2;
    }

    void build_mask() {
        int current_mask = 0;
        // We use this mask so that we only keep track of the at most
        // the size of the block bits
        int discard_mask = (1 << blockSize) - 1;
        for (int i = 0; i < n; ++i) {
            current_mask = (current_mask << 1) & discard_mask;

            while (current_mask > 0 &&
                   op(i, i - msb_index(lsb(current_mask))) == i) {
                // Current value is smaller than the value represented by the
                // last 1 in curr_mask, so that bit should be off
                current_mask ^= lsb(current_mask);
            }
            current_mask |= 1;
            mask[i] = current_mask;
        }
    }
    void build_sparse_table() {
        // Fill first row
        for (int i = 0; i < rowSize; ++i) {
            sparseTable[i] = small_query_index(blockSize * (i + 1) - 1);
        }
        for (int j = 1; (1 << j) <= rowSize; ++j) {
            for (int i = 0; i + (1 << j) <= rowSize; ++i) {
```

```
                int previousStep = 1 << (j - 1);
                sparseTable[rowSize * j + i] =
                    op(sparseTable[rowSize * (j - 1) + i],
                       sparseTable[rowSize * (j - 1) + i + previousStep]);
            }
        }
    }
};
```

1.3 Fenwick tree (BIT)

A Fenwick tree or binary indexed tree (BIT) can be used to calculate the value of a **reversible** function F over a range of values. Given an array of elements A of size N:

- Given a range $[l, r]$, the value $F(a_l, \ldots, a_r)$ can be calculated in $\boldsymbol{\mathcal{O}(\log n)}$
- Updating one of the values of the array takes $\boldsymbol{\mathcal{O}(\log n)}$
- Requires the same amout of memory as the array A

To do so, we create a new array `ft` where each element represents the value of the function over a particular range. Let LSOne(i) be the result of setting to 0 all bits of i except for the least significant one. For instance, LSOne(0110) = 0010. Then, the i-th element of `ft` contains the value $F(a_{i-\text{LSOne}(i)+1}, \ldots, a_i)$.

In most cases, F will be the sum of all elements $F(a_l, \ldots, a_r) = \sum_l^r a_i$. In that case, ft[i] contains the sum of all the elements from position $i - \text{LSOne}(i)$ (excluded) to position i (included).

It is also important to note that this BIT is **1-indexed interally** to simplify slightly the implementation. However, the interface automatically subtracts one to all index parameters given.

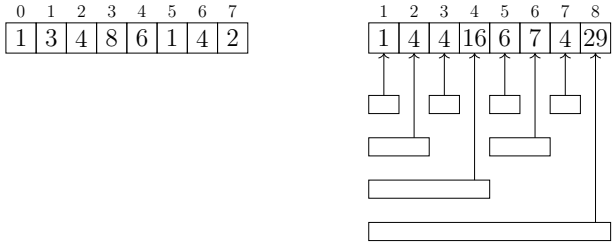
Range Update and Point Query

The standard BIT allows point update and range query. Using a difference array, we can transform them into range update and point query. If B is the difference array of A, the elements of B are defined as: $b_i = a_i - a_{i-1}$ and $b_1 = a_1$. Therefore:

- $\sum_1^i b_j = a_i$, converting the range query into a point query
- To update the elements in the range $[l, r]$ by d, we can add d to the l-th element and subtract d from the r-th element. In doing so, queries that reach until a point between l and r will be increased by d but queries that reach further will not be affected. Therefore, we have converted two point updates into a range update.

Example

For instance, if we use the addition operation, array on the left would be processed into the BIT tree displayed on the right. Each box under the BIT tree represents the elements that are added to get the value of that position.



```cpp
// Return lsb
int LSOne(int a) {
    return a & (-a);
}

template <typename T>
class FenwickTree {
private:
    vector<T> ft;

public:
    FenwickTree(const vector<T> &data)
        : ft(data.size() + 1, 0) {
        for (int i = 0; i < data.size(); ++i) {
            adjust(i, data[i]);
        }
    }

    //RSQ in range [0, index]
    T rsq(int index) {
        index++;
        T sum = 0;
        while (index != 0) {
            sum += ft[index];
            index -= LSOne(index); // Delete the lsb
        }
        return sum;
    }
    //RSQ in range [begin, end]
    T rsq(int begin, int end) {
        return begin == 0 ? rsq(end) : rsq(end) - rsq(begin - 1);
    }

    // Add the delta to the index element
    void adjust(int index, T delta) {
        index++;
        while (index < ft.size()) {
            ft[index] += delta;
            index += LSOne(index);
        }
    }
};
```

1.4 Segment Trees

A Segment Tree is a data structure that allows answering range queries over a range effectively while permitting modifications to be performed to the elements in the array. Furthermore, a Segment Tree can answer many types of queries. In fact, we only need that the operation satisfies the following property:

The value of a range can be calculated from the values of two disjoint subranges whose union is the entire range.

Most common operations such as the maximum, the minimum or the gcd all fulfill this property. For instance, in the case of the sum:

$$i < k < j \implies \text{sum}(a_i, \ldots, a_j) = \text{sum}(a_i, \ldots, a_k) + \text{sum}(a_{k+1}, \ldots, a_j)$$

However, sometimes it may be necessary to store additional information in the vertices of the Segment Tree to be able to construct the solution from the data from the two subranges. The time complexities are the following:

- Build time: $\boldsymbol{\mathcal{O}(n \log n)}$
- Query: $\boldsymbol{\mathcal{O}(\log n)}$
- Update $\boldsymbol{\mathcal{O}(\log n)}$

Example problem

We are given a string s consisting only of 0s and 1s. We must be able to perform the following types of queries:

- Calculate the length of the maximum continuous substring of 1s inside a given interval
- Flip all numbers in an interval. That is to say, all 1s turn into 0s and the other way around.

In this case, we need to use lazy propagation and store the following information in each node:

```
struct node{
    int prefix1s, sufix1s, prefix0s, sufix1s;
    int longestInterval0, longestInterval1;
    int lazyFlip;
};
```

Storing the prefix and suffix information allows us to calculate the value of intervals that begin in the elements corresponding to the left child and end in elements that correspond to the right child. Therefore, we will be able to calculate the answer of each node looking only at their children.

1.4.1 Lazy propagation

Lazy propagation enables the Segment Tree to perform updates over a range in logarithmic time. However, in order to be able to use this technique we need the following property:

Two updates that affect the same node can be merged into a single update

As with the basic property of Segment Trees, this one also holds true for the most common operations. In fact, as long as there is only one type of update query, it will generally be true. However, in scenarios where different updates may be requested, it is critical that they can be merged.

Example

Problem `UVa-11402` boils down to performing the following three operations over ranges of a binary string and then counting the number of 1s in any range. Otherwise, the number of nodes modified by a single query would be $\mathcal{O}(n)$, which would ruin the logarithmic time complexity.

1. Set all positions in the range to 1
2. Set all positions in the range to 0
3. Flip all positions in the range

In this case, when propagate the lazy value stored in a node to their children, we must "merge" it with the lazy value already stored in the children. For instance:

$$\texttt{lazy[node] = 1 \& lazy[child] = 2} \implies \texttt{lazy[child] = 1}$$

Lazy lazy propagation

In some cases, we can avoid the push down operation. Instead, we can store the lazy values in the nodes that they are stored into during the usual update procedure and collect the values on the way up during the query process. For instance, to implement range updates and point queries, in an initially empty array, we might use the following code:

```
28 void _update(int current, int l, int r) {
29     if (l > qRight || r < qLeft) {
30         return;
31     }
32     if (l ⩾ qLeft && r ⩽ qRight) {
33         tree[current] += qDelta;
34         return;
35     }
36     int mid = (l + r) >> 1;
37     _update(current << 1, l, mid);
38     _update(current << 1 | 1, mid + 1, r);
39 }
```

```
40
41  ll _query(int current, int l, int r) {
42      if (l == r) {
43          return tree[current];
44      }
45      int mid = (l + r) >> 1;
46      if (qLeft <= mid) {
47          return tree[current] + _query(current << 1, l, mid);
48      } else {
49          return tree[current] + _query(current << 1 | 1, mid + 1, r);
50      }
51  }
```

However, this technique can only be used in very simple cases as we can very easily run into timing issues when the effect of the queries depends on the previous queries.

Nevertheless, when it can be used, it reduces a lot of the overhead associated with range queries in the segment tree.

1.4.2 Range update and query

Even though the implementation of a Segment Tree highly depends on the nature of the update queries, the following implementation should at least serve as a starting point in many situations.

The template parameters have the following meaning:

- `T`. Type of the elements that are stored in every node. Note that this could be a structure with many fields as we saw in the first example problem of this section.

  It is also important to note that if a `struct` is used, an equality operator must be defined.

- `merge`. This function takes the values of the children of a given node and returns the value of that node.

  Note that in this implementation, this same function is used to merge the propagation values. This will work for simple problems, but more complex ones will need a separate function to merge these.

- `initialize`. This function takes the position of the element and returns the initial value of that element. In most cases, it will simply contain an array look-up. However, it can also initialize all positions to a constant value or to any value that can be calculated from the position.

- `neutral`. This is a `constexpr` value that stores an element that will be ignored by the merge function. For instance, if the operation was the sum, this value would be 0 and if the operation was the minimum, this value could be $+\infty$.

```cpp
template <typename T, T (*merge)(T, T), T (*initialize)(int), const T &neutral>
class SegTree {
    int n;
    int ql, qr;
    T qValue;
    vector<T> tree;
    vector<T> lazy;

    void _build(int current, int l, int r) {
        lazy[current] = nil;
        if (l == r) {
            tree[current] = initialize(l);
        } else {
            int mid = (l + r) >> 1;
            _build(current << 1, l, mid);
            _build(current << 1 | 1, mid + 1, r);
            tree[current] = merge(tree[current << 1], tree[current << 1 | 1]);
        }
    }
    void push_down(int current) {
        if (lazy[current] == neutral) {
            return;
        }
        tree[current << 1] = merge(tree[current << 1], lazy[current]);
        tree[current << 1 | 1] = merge(tree[current << 1 | 1], lazy[current]);
        lazy[current << 1] = merge(lazy[current << 1], lazy[current]);
        lazy[current << 1 | 1] = merge(lazy[current << 1 | 1], lazy[current]);
        lazy[current] = neutral;
    }
    T _query(int current, int l, int r) {
        if (l > qr || r < ql) {
            return neutral;
        }
        if (l ⩾ ql && r ⩽ qr) {
            return tree[current];
        }
        push_down(current);
        int mid = (l + r) >> 1;
        T n1 = _query(current << 1, l, mid);
        T n2 = _query(current << 1 | 1, mid + 1, r);
        return merge(n1, n2);
    }

    void _update(int current, int l, int r) {
        if (l > qr || r < ql) {
            return;
        }
```

```
        if (l >= ql && r <= qr) {
            tree[current] = merge(tree[current], qValue);
            lazy[current] = merge(lazy[current], qValue);
            return;
        }
        push_down(current);
        int mid = (l + r) >> 1;
        _update(current << 1, l, mid);
        _update(current << 1 | 1, mid + 1, r);
        tree[current] = merge(tree[current << 1], tree[current << 1 | 1]);
    }

public:
    SegTree(int actualSize) : n(actualSize), tree(4 * n), lazy(4 * n) {
        _build(1, 0, n - 1);
    }
    T query(int l, int r) {
        ql = l;
        qr = r;
        return _query(1, 0, n - 1);
    }
    void update(int l, int r, T value) {
        ql = l;
        qr = r;
        qValue = value;
        _update(1, 0, n - 1);
    }
};
```

1.4.3 Persistent segment trees

In some scenarios it is useful to be able to execute a query or an update on a previous state of the segment tree. To implement this feature in our segment tree, we could create a new segment tree every time that we execute an update. However, if we were to answer q queries with this approach, the complexity be $\mathcal{O}(q\,n\log n)$. However, by using persistent segment trees, we can bring the complexity down to $\mathcal{O}(q\log n)$ while having full access to all previous states of the segment tree.

In order to do so, we have to observe that each query will at most modify $\log n$ nodes. Therefore, we can create new versions for those nodes while keeping the rest of the nodes. The simplest way to implement this concept is with a pointer-based implementation of the segment tree.

Implementation

The following implementation calculates range sums and allows point updates. The first element of the implementation is the `node` structure. This represents a single node in the persistent segment tree and can belong to different versions of the segment tree. Therefore, we will allocate it in the heap and leave it there for all versions to access.

```
20 struct node {
21     node *l = nullptr, *r = nullptr;
22     ll sum;
23     node(ll val) : sum(val) {}
24     node(node *l, node *r) : l(l), r(r), sum(0) {
25         sum = (l ? l->sum : 0) + (r ? r->sum : 0);
26     }
27 };
```

The rest of the implementation can be split into two parts. The first one is the "private" one, which deals with the nodes itself while the second part is the interface that should be used to interact with the segment tree. Nevertheless, no members are private since in some scenarios (walking down the tree) we might need to access the internal members.

Regarding the private section, we have three functions:

- `build` creates a segment tree initialized to 0 and returns a pointer to the root node
- `_query` is completely analogous to the normal query function in a segment tree
- `_update` performs the same action as the "traditional" update function. However, it returns a pointer to the new node that was created with the updated value. We can see in line 63 how the newly created nodes partially point to older nodes in the segment trees (those whose value was not modified by the query)

```cpp
struct PersistSegTree {
    static int ql, qr;
    static int pos, new_val;

    node *head = nullptr;
    int n;

    node *build(int l, int r) {
        if (l == r) {
            return new node(0);
        } else {
            int mid = (l + r) >> 1;
            return new node(build(l, mid), build(mid + 1, r));
        }
    }

    ll _query(node *current, int l, int r) {
        if (l > qr || r < ql) {
            return 0;
        }
        if (l >= ql && r <= qr) {
            return current->sum;
        }
        int mid = (l + r) >> 1;
        return _query(current->l, l, mid) + _query(current->r, mid + 1, r);
    }

    node *_update(node *current, int l, int r) {
        if (l == r) {
            return new node(new_val + current->sum);
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) {
            node *n_l = _update(current->l, l, mid);
            return new node(n_l, current->r);
        } else {
            node *n_r = _update(current->r, mid + 1, r);
            return new node(current->l, n_r);
        }
    }

```

Finally, the interface of the persistent segment tree only has two operations, it can either query a range or perform an update, which would return a new `PersistentSegTree`. In both cases, the functions only set the right internal variables and call the private ones.

It is important to remember the declaration on line 87 of the static variables used internally by the segment tree. These must be declared outside the class.

```
73      ll query(int left, int right) {
74          ql = left;
75          qr = right;
76          return _query(this→head, 0, n - 1);
77      }
78      PersistSegTree update(int position, int increment) {
79          pos = position;
80          new_val = increment;
81          PersistSegTree res;
82          res.n = n;
83          res.head = _update(this→head, 0, n - 1);
84          return res;
85      }
86  };
87  int PersistSegTree::ql, PersistSegTree::qr, PersistSegTree::pos,
88      PersistSegTree::new_val;
89
```

Example

As an example, in problem `SPOJ-KQUERY`, the creation of the versions of the segment tree is performed as follows:

```
1  vector<PersistSegTree> versions;
2  versions.push_back(PersistSegTree(n));
3  for (int i = 0; i < n; ++i) {
4      int value = getCompressed[originalData[i]];
5      versions.push_back(versions.back().update(value, 1));
6  }
```

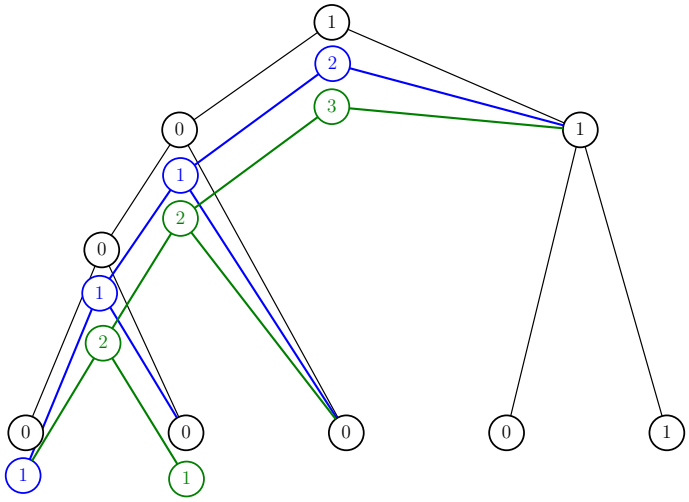
And, afterwards, we can query a specific version i of the segment tree between a and b by using:

```
1  int ans = versions[i].query(a,b);
```

Example

Let us consider how the range sum persistent segment tree would behave during some queries. In this example, we will start by constructing a persistent segment tree over an array `a = [0, 0, 0, 0, 0, 1]`. This will result on the segment tree in black. Now, we will perform the following queries:

- *Add 1 to the first position.* To do so, we have to change all ancestors of the first leaf. This process will lead to the segment tree in blue. The key observation here is the fact that we have only added 4 blue nodes and some of them are connected to black nodes. In doing so, we can reuse nodes that have not been modified.
- *Add 1 to the second position.* This will result in the segment tree in green, which has nodes connected to both black and green node. Again, this enables us to change only $\mathcal{O}(\log n)$ nodes in each iteration.



As we can see, if we keep track of the roots that are created in each update, we can access any version of the segment tree.

Number of elements less or equal than k in a range

An example of this problem is SPOJ-KQUERY In order to solve this problem we will build a histogram of every prefix of the array. Given the example array a, the histograms of the different prefixes are the rows of the table. Now, we can build a persistent segment tree for

a =

0	1	2	3	4	5
5	1	2	5	2	3

Length	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	1
3	1	1	0	0	1
4	1	1	0	0	2
5	1	2	0	0	2
6	1	2	1	0	2

those histograms. The first version of the segment tree will contain the histogram of the empty prefix, the second version will contain the histogram of the prefix of length one…. Since each histogram only differs from the previous one in one value, the persistent segment tree will only have $n + 1$ version. To create these versions, we could execute:

```
vector<PersistSegTree> versions;
versions.push_back(PersistSegTree(n)); // Empty histogram
for (int i = 0; i < n; ++i){
    versions.push_back(versions.back().update(a[i], 1));
}
```

Now, if we are given a range $[l, r]$ (1-indexed) and asked to return the number of elements less or equal to k in that range, we can calculate it in the following way:

```
int ans = versions[l].query(0,k) - versions[r-1].query(0,k);
```

Find the k-th smallest element in a range (Slow)

Create a segment tree of size n and iterate through the elements in the array sorted by value. At each iteration, set the positions corresponding to that element to 1.

Then, when asked a query in a range we just have to know which is the first version that has at least k positions set to 1 in that range. This can easily be accomplished with a binary search over the segment trees. However, this binary search introduces another log factor in the complexity, which gives the solution an $\mathcal{O}(q \log^2 n)$ time complexity to answer q queries.

Find the k-th smallest element in a range (Fast)

Now we will solve the same problem but with a time complexity of $\mathcal{O}(q \log n)$. To do so, we will use a similar approach as in the first application of the persistent segment tree and build a histogram for every prefix. When we get a query for the range $[l, r]$, we can walk down the segment tree . In each node, we will check whether the answer is in the left or the right child. There are two options:

- The sum of the left child is greater or equal to the target value. In this case, the answer is one of the indices of the left child since we are looking for the first index whose sum is at least k.
- The sum of the left child is less than the target value. Then the solution is one of the indices of the right child. However, before continuing the search, we have to update the target value and subtract from it the sum of the left child. This is because once we descend to the right subtree, the nodes that we will encounter only contain the sum up to the end of the right subtree.

Finally, we have to take into account that we do not have a specific segment tree for the range $[l, r]$ but it is easy to see that the values in this segment tree would be the difference between the two segment trees in the edges of the range. Therefore, to calculate the answer:

```
90  int findKth(node *nl, node *nr, int l, int r, int target) {
91      if (l == r) {
92          return l;
93      }
94      int leftSum = nr->l->sum - nl->l->sum;
95      int mid = (l + r) / 2;
96      if (leftSum >= target) {
97          // Going left
98          return findKth(nl->l, nr->l, l, mid, target);
99      } else {
100         // Going right
101         return findKth(nl->r, nr->r, mid + 1, r, target - leftSum);
102     }
103 }
```

```
1  // In main():
2  int res = findKth(histograms[l-1].head, histograms[r].head, 0, size - 1, k);
```

An example of this problem is `SPOJ-MKTHNUM`

Chapter 2

Graphs

2.1 Dijkstra's

Shortest path from `orig` node to `dest` (or to every node) in a graph that does not contain negative edges. It chooses the best path greedily in each iteration and, therefore, it only works on graphs without negative weights.

```
20 int dijkstra(int orig, int dest, vector<vector<pii>> &graph) {
21     vi dists(graph.size(), INT_MAX / 10);
22     priority_queue<pii, vector<pii>, greater<pii>> q;
23
24     q.push({0, orig});
25     dists[orig] = 0;
26     while (!q.empty()) {
27         auto a = q.top();
28         q.pop();
29         if (a.se == dest) { // If we already reached the destination
30             return a.fi;
31         }
32         if (dists[a.se] != a.fi) { // If we have found a better option
33             continue;
34         }
35         for (auto b : graph[a.se]) { // Else iterate through the edges
36             if (dists[b.fi] > dists[a.se] + b.se) {
37                 dists[b.fi] = dists[a.se] + b.se;
38                 q.push({dists[b.fi], b.fi});
39             }
40         }
41     }
42     return dists[dest];
43 }
```

Running time: $\mathcal{O}((E + V) \log V)$
(V = vertices, E = edges)

Remarks

- If we ignore the check in line 30, we can return the distances vector, which will contain the shortest distance from `dist` to every other node.
- If we are doing some kind of pruning it is imperative that we prune as many branches as possible in the main loop. That is to say, we should introduce as many `if` statements in line 36 to make sure that we run the `for` loop as few times as possible.

  An example of this approach is problem `UVA-11635`. In that problem, we add a lot of branches to the queue (we may run the `for` loop twice in some nodes) but we prune them in the main loop. Thus the running time is still acceptable.
- However, pruning the branches with a higher cost can be quite complicated when we have to optimize several factors (see section 2.1.1). Therefore, we will omit the check in line `33` if we think it might discard relevant options.

2.1.1 Dijkstra's with cost and distance

In this problem we are going to define two values for each edge, cost and distance:

- *Cost*. This is the number that we want to minimize. It is the equivalent to the usual cost in a normal Dijkstra problem
- *Distance*. We define the distance of a path as the sum of the distances of the edges in it. And the distance of the path from source to destination cannot exceed a given limit (`B`)

This problem can be solved with a slightly modified version of Dijkstra's:

```
20 int B;
21 using pipii = pair<int, pii>;
22 vector<vector<pipii>> adyList; //destination, cost, distance
23 ll dijkstra(int orig, int dest) {
24     vi distances(adyList.size(), iinf);
25     priority_queue<pipii, vector<pipii>, greater<pipii>> q; //cost,node,dist
26 
27     distances[orig] = 0;
28     q.push({orig, {0, 0}});
29 
30     while (!q.empty()) {
31         auto a = q.top();
32         q.pop();
33 
34         if (a.se.fi == dest) {
35             return a.fi;
36         }
37         distances[a.se.fi] = a.se.se;
38 
39         for (auto b : adyList[a.se.fi]) {
40             if ((a.se.se + b.se.se < distances[b.fi]) ) {
41                 q.push({a.fi + b.se.fi, {b.fi, a.se.se + b.se.se}});
42             }
43         }
44     }
45     return -1;
46 }
```

Running time: $\mathcal{O}\big(B \cdot (V + E \log E)\big)$
(V = vertices, E = edges, B = max_distance)

In this algorithm, we will process each node at most `B` times. Furthermore, if we process a node with a cost c and a distance d, we know that this is the best possible cost for that distance d since all the nodes in the queue have a higher cost already. Therefore, we can close the node for that particular distance value (line 37).

An example of this algorithm is `SWERC-19_20-A`

2.2 Bellman Ford's

Shortest parth from `orig` to every other node. It is slower than Dijkstra but it works on graphs with negative weights.

This algorithm works by trying to relax every edge $V-1$ times. If there are no negative cycles, after $V-1$ iterations, we must have found the minimum distance to every node. Therefore if after these iterations, we run another iteration and the distance to a node decreases, we must have a negative cycle.

```
20 vi bellmanFord(int orig, vector<pair<pii, int>> edges, int n) {
21     vi dist(n, INT_MAX / 10);
22     dist[orig] = 0;
23     for (int i = 0; i < n - 1; ++i) {
24         for (auto e : edges) {
25             dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
26         }
27     }
28     // dist[i] contains the shortest path from 0 to i
29 
30     //We can now check for a negative cycle
31     bool negativeCycle = false;
32     for (auto e : edges) {
33         if (dist[e.fi.se] > min(dist[e.fi.se], dist[e.fi.fi] + e.se)) {
34             negativeCycle = true;
35         }
36     }
37     return dist;
38 }
```

Running time: $\mathcal{O}(VE)$
(V = vertices, E = edges)

Remarks

- If we keep track of the distance that decrease when we check for a negative cycle, we will get at least one node of every negative cycle present in the graph.

  We can use this, for instance, to check if we can reach a node with a cost smaller than a given bound. If it is connected to a node in a negative cycle, it's distance will be as small as we want it to be (by looping in the cycle).

  This can be seen at play in `UVA-10557`

- If we modify slightly the main loop, iteration i will be the result of considering paths of at most $i+1$ edges:

```
for (int i = 0; i < n - 1; ++i) {
        for (auto e : edges) {
            dists2[e.fi.se] = min(dists2[e.fi.se], dists[e.fi.fi] + e.se);
        }
    dists = dists2;
}
```

This can be seen at play in `UVA-11280`

2.3 Warshall's

Warshall's algorithm solves the APSP (All pairs shortest path) using DP. Each iteration of the outer loop tries to add a node (the k-th node in particular) to the path between all pairs of nodes. We can think of the comparison as:

Is the path from i to j shorter if we first move from i to k and, then, from k to j?

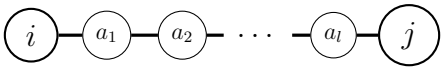
It is important to note that we must use an adjacency matrix in this implementation.

```
void warshall(int n) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                adyMat[i][j] = min(adyMat[i][j], adyMat[i][k] + adyMat[k][j]);
            }
        }
    }
}
```

Running time: $\mathcal{O}(V^3)$
(V = vertices, E = edges)

Obtaining the path

To get the explicit path, we will store the last vertex in the path that the algorithm found to go from i to j. That is to say, after a given iteration the optimal path from i to j is: Then



we should store that the last node in the path that goes from i to j is a_j. The code to do so is the following:

```cpp
ll lastInPath[lim][lim];

void warshall(int n) {
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (adyList[i][j] > adyList[i][k] + adyList[k][j]) {
                    adyList[i][j] = adyList[i][k] + adyList[k][j];
                    lastInPath[i][j] = lastInPath[k][j];
                }
            }
        }
    }
}

// Give an empty vector
void getPath(int i, int j, vector<int> &path) {
    if (i != j) {
        getPath(i, lastInPath[i][j], path);
        path.push_back(j);
    }
}
```

2.4 DFS

2.4.1 Articulation points and bridges

These algorightms can be used in undirected graphs, and we will use the following definitions:

- **Articulation point**. A node whose removal would increase the number of connected components of the graph. That is to say that it "splits" a connected component.
- **Bridge**. An edge whose removal increases the number of connected components in the graph.

We will use a modified version of DFS to solve this problem. We mainly introduce two new properties for every node:

- `num`. Time at which the node was first explored by DFS
- `low`. Earliest node that can be found in the DFS spanning tree that starts from this node

When we visit a node, for every edge, there are two options:

- **Tree edge**. This edge point to a node that has not been discovered yet. As such we explore it (calling `dfs`) and, we update the value of `low` for the current node.

  After the update, we can now process the child since we will not visit it again and it's DFS tree has been fully explored
- **Back edge**. This edge points to a node that has already been visited. Therefore, it will have a relatively low `num` and we use it to update the `low` value of the current node

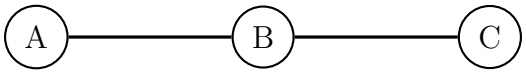
```cpp
vector<vi> adyList; // Graph
vi num, low;        // num and low for DFS
int cnt;            // Counter for DFS
int root, rchild;   // Root and number of (DFS) children
vi artic;           // Contains the articulation points at the end
set<pii> bridges;   // Contains the bridges at the end

void dfs(int nparent, int nnode) {
    num[nnode] = low[nnode] = cnt++;
    rchild += (nparent == root);

    for (auto a : adyList[nnode]) {
        if (num[a] == -1) { // Tree edge
            dfs(nnode, a);
            low[nnode] = min(low[nnode], low[a]);

            if (low[a] >= num[nnode]) {
                artic[nnode] = true;
            }

            if (low[a] > num[nnode]) {
                bridges.insert((nnode < a) ? mp(nnode, a) : mp(a, nnode));
            }
        } else if (a != nparent) { // Back edge
            low[nnode] = min(low[nnode], num[a]);
        }
    }
}
void findArticulations(int n) {
    cnt = 0;
    low = num = vi(n, -1);
    artic = vi(n, 0);
    bridges.clear();

    for (int i = 0; i < n; ++i) {
        if (num[i] != -1) {
            continue;
        }
        root = i;
        rchild = 0;
        dfs(-1, i);
        artic[root] = rchild > 1; //Special case
    }
}
```

Running time: $\mathcal{O}(V + E)$

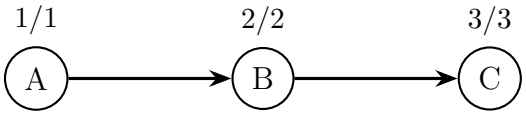
(V = vertices, E = edges)

Explanation

The first graph that we will consider is the following: After applying DFS on A we get the



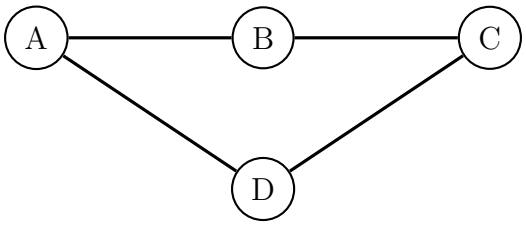
following DFS spanning tree. Above every node, we have included `num / low`.



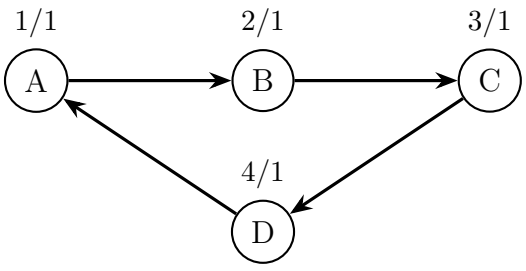
This is a rather simple graph and it's only articulation point is B. This is because it has a child whose `low` value is greater than or equal to B's `num` value. Therefore, there is no connection from that child (namely C) to a node explored before B that does not go through B.

However, we can already see that the root has to be treated as a separate case. The root will be an articulation point iff it has more than one child in its DFS tree. It is important to note that the children of the DFS tree need not be the same as the children of the root in the initial graph.

Let's look at another example



As before, we show the DFS spanning tree:



Now we have no articulation point since there is no node that has a child with a `low` greater than or equal than the parent's `num`. This difference is caused by the fact that now there is an edge that makes C accessible through a path that does not involve traversing B.

In this case, we can see how the root has only one child in the spanning tree but two in the initial graph. This reflects the fact that those numbers need not coincide.

Finally, the condition for a bridge is: `low[child]>num[parent]`. This is equivalent to stating that the child has no other way of reaching either the parent or a node that was explored before the parent

2.4.2 Trajan's algorithm for strongly connected components

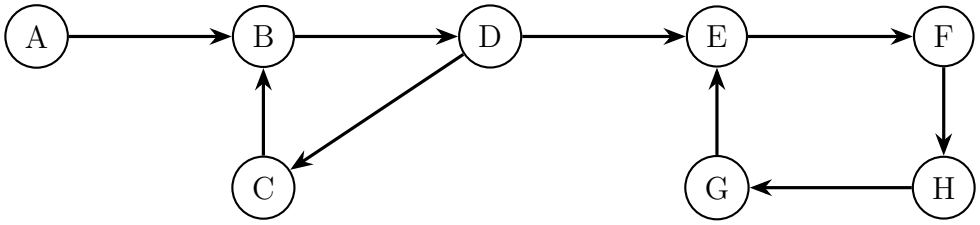
In a directed graph, we say that a subset of vertices comprises a strongly connected component if every vertex is reachable from every other vertex in this subset.

We will now present an algorithm that divides the graph into strongly connected parts that are as large as possible. It will use some of the same concepts as in the previous section.

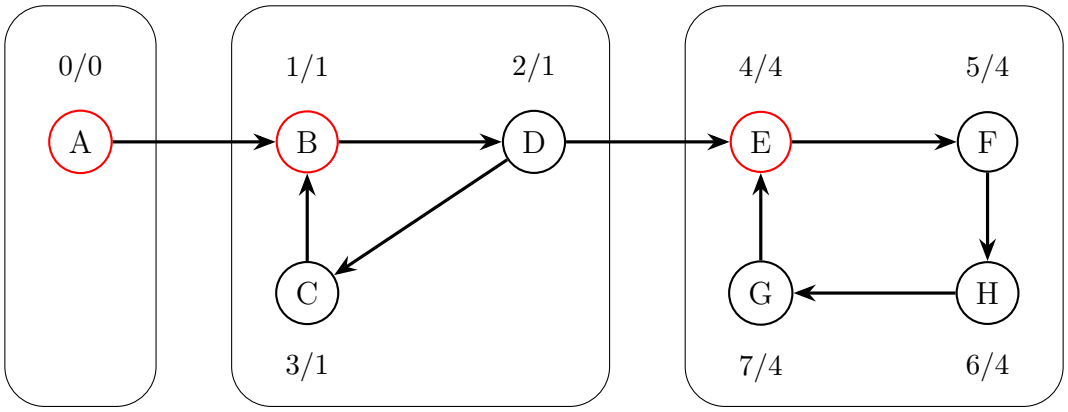
Explanation

The main idea is the following: if when we are done with a node, we have not been able to reach a node that was further back, this one is the root of a SCC.

Let's explain this reasoning with an example:



And, after running DFS, we would get the following spanning tree with three SCCs.



Let's look at the moment in which we close node F. As we can see the `low` value is lower than the `num`. Therefore, there is a connection from F to another node that was visited before and it will be a part of the SCC "generated" by that node.

However, when we close E, we can see that the `low` and the `num` are equal. As a result, there is no way to get to a node that has a number lower than E's through the spanning tree of E. Therefore, there is no way to have a bigger SCC that contains E.

To get the nodes that are part of the SCC, we need to get all the nodes in the stack before the one that "generates" the SCC (g). This is because the stack only contains the

nodes that belong to the spanning tree of g and do not belong to any other SCC. Therefore, they must belong to the one generated by g.

Furthermore, all those nodes will have the same `low` value since the `low` any node with a higher value must have been processed before. Therefore, g is accessible from all those nodes and, clearly, all those nodes are accessible from g. Thus, they fulfill the definition of an SCC.

```cpp
vector<vi> adyList; // Graph
vi low, num;        // num and low for DF
int cnt;            // Counter for DFS
stack<int> st;
vi inStack;      // Position in the stack + 1
vector<vi> sccs; // Contains the SCCs at the end

void dfs(int u) {
    low[u] = num[u] = cnt++;
    st.push(u);
    inStack[u] = st.size();
    for (auto a : adyList[u]) {
        if (num[a] == -1) {
            dfs(a);
        }
        if (inStack[a]) {
            low[u] = min(low[u], low[a]);
        }
    }
    if (low[u] == num[u]) { // Root of a SCC
        vi v;
        // Add all the nodes till u (included)
        int lim = inStack[u];
        while (st.size() && st.size() >= lim) {
            v.push_back(st.top());
            inStack[st.top()] = false;
            st.pop();
        }
        sccs.push_back(v);
    }
}
void TarjanSCC(int n) {
    low = num = vi(n, -1);
    inStack = vi(n, 0);
    st = stack<int>();
    cnt = 0;
    sccs = vector<vi>();

    for (int i = 0; i < n; ++i) {
        if (num[i] == -1) {
            dfs(i);
        }
    }
}
```

Running time: $\mathcal{O}(V + E)$

(V = vertices, E = edges)

2.4.3 Find all cycles in a directed graph with out degree 1

If every node in the graph has an outdegree of 1, DFS can find all cycles in a directed graph by keeping track of the nodes that are currently in the stack. Those are the nodes that DFS has entered but has not exited yet. When the current node is connected to a node in the stack, we have found a cycle.

The implementation would look like this:

```
set<int> presentInLoop;
void dfs(int cnode) {
    inStack[cnode] = true;
    visited[cnode] = true;
    int ne = nextt[0][cnode];
    if (inStack[ne]) {
        presentInLoop.insert(ne);
    } else if (!visited[ne]) {
        dfs(ne);
    }
    inStack[cnode] = false;
}
```

Even though this might seem like a very specific scenario. All graphs that are created from functions will fulfill that the outdegree of all vertices is 1. Therefore, we will be able to use this algorithm.

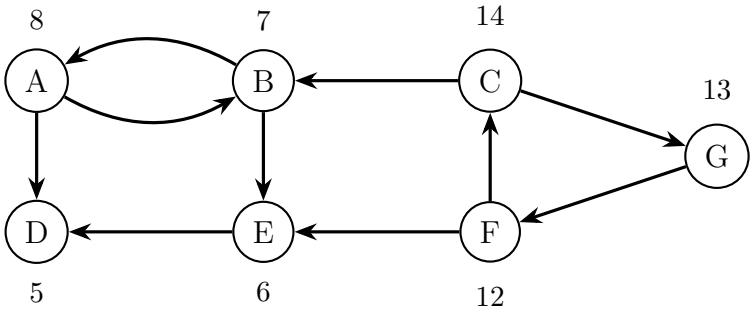
2.5 Kosaraju's

Kosaraju's algorithm is a slightly simpler method for finding the SCC of a graph. On the other hand, it is also somewhat slower than Trajan's approach since it will require running DFS on the graph twice. The algorithm follow this procedure:

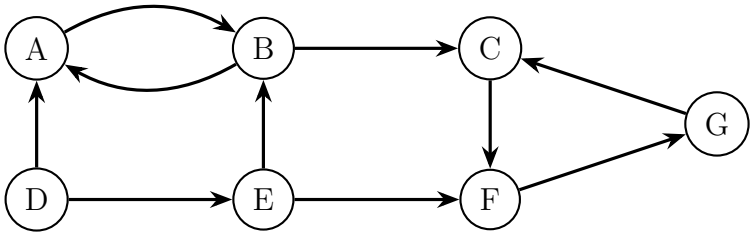
1. Run DFS on the graph and store the vertices in postorder in a list called `postorder`.
2. Reverse the graph
3. Loop through the nodes in `postorder` starting from the one that was closed the last. For each of them check if it has already been added to a SCC (visited). If it hasn't, run DFS from that node and create a SCC with all the nodes that this DFS visits and hadn't been visited before.

Example

Let's look at an example graph where we have run DFS starting first in node A and, since it didn't explore the entire graph, we also run it from node C. We have included the time at which each node was closed:



Now we reverse the graph and we get the following:

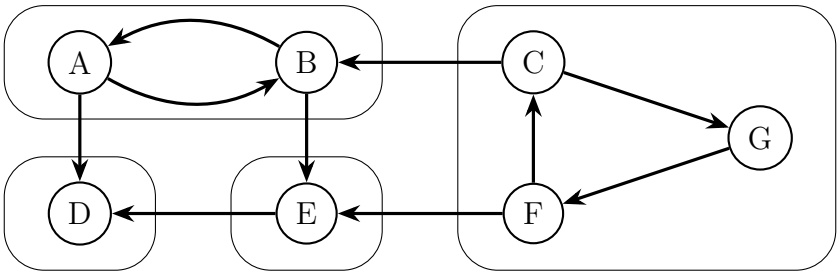


Finally, we can loop through the nodes in postorder, starting with the node that was close the last:

$$\text{postorder} = \{C, G, F, A, B, E, D\}$$

1. We run DFS on node C and we get the SCC:
$$s_1 = \{C, F, G\}$$
2. We skip nodes G and F in the list since they are already in a SCC and we run DFS on node A, getting the SCC:
$$s_2 = \{A, B\}$$
3. We skip node B since it is already in a SCC and we run DFS on node E, getting the SCC:
$$s_3 = \{E\}$$
4. We finally run DFS on node D, and we get the last SCC:
$$s_4 = \{D\}$$

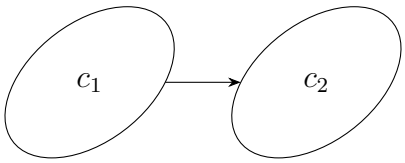
Now, returning to the initial graph, we have found the following SCCs:



Explanation

Let's look at why this algorithm works.

Firstly, we have to take into account that the SCCs of a graph G are preserved when we reverse all the edges and get G^t. The only relevant issue is the order in which we process the SCCs. Let's assume we have a graph that has two SCC's: We have three options:



- *They are not connected.* In this case, it does not matter whether we explore one or the other first since DFS will not "leak" from one of them to the other
- *They are connected only in one direction.* This is the case that the figure shows and it is the most important one. If we explore c_2 before exploring c_1, DFS will first explore the entirety of c_2 and then, when it is exploring c_1 it will not "leak" to c_2 because those nodes are already marked as visited.
- *They are connected in both directions.* This can never happen since, if they were to be connected bidirectionally, they would form a single SCC, not two.

Therefore, the correctness of this algorithm simply depends on exploring the SCCs in the right order.

Let's demonstrate the following claim:

In the previous setting, the maximum closing time of the nodes in c_1 will be greater than the maximum closing time of the nodes in c_2

We just have to distinguish two cases:

- If we started exploring c_2 before c_1, there is no way to get to c_1 from c_2. Therefore, we will start exploring c_1 when we have already closed c_2, which means that all nodes in c_2 will be closed before the nodes in c_1 are even "opened".
- If we started exploring c_1 before c_2, at some point, DFS will leak to c_2 and it will explore it entirely before returning to c_1. Therefore, the closing time of every node in c_2 will be lower than the closing time of the node where DFS was started in c_1.

This completes the proof of that statement. Let's now apply it to check the correctness of this algorithm using induction.

- *Base case:*

  After reversing the graph, we start with the node that was closed the last (a). Let's call it's SCC s_1. Let's assume that there is an edge from s_1 to another SCC (s_α) in G^t, which would make DFS leak to that SCC. However, if that was the case, there would be an edge from s_α to s_1 in G and, therefore, we can apply the previous claim.

  In that scenario, the maximum closing time of s_α would be greater than the maximum closing time of s_1. This is a contradiction because we have stated that $a \in s_1$ is the node that was closed the last.

  Therefore, there cannot be any edge from s_1 to another SCC.

- *Inductive step:*

  Let's now assume that we have already processed n SCCs. We now choose a node b, which is the node with the highest closing time such that $b \notin s_i,\ i = 1, \dots n$.

  We will call the SCC of this node s_{n+1}. As before, we want to prove that s_{n+1} is not connected to any s_β that hasn't been explored yet in G^t.

  We will assume that there is an edge from s_{n+1} to a $s_\beta \notin \{s_1 \dots s_n\}$. Therefore, in G, there is an edge from s_β to s_{n+1}, which implies that the maximum closing time of s_β is higher than the maximum closing time of s_{n+1}.

  Let's define $x_\beta :=$(the node with the maximum closing time of s_β). We just stated that the closing time of x_β is greater than the closing time of b (there cannot be a node with a greater closing time in s_{n+1}).

  However, this is a contradiction. Therefore, s_{n+1} cannot be connected to an SCC that has not been explored and, thus, DFS will not leak.

□

```cpp
vector<vi> adyList;  // Graph
vector<int> visited; // Visited for DFS
vector<vi> sccs;     // Contains the SCCs at the end

void dfs(int nnode, vector<int> &v, vector<vi> &adyList) {
    if (visited[nnode]) {
        return;
    }
    visited[nnode] = true;
    for (auto a : adyList[nnode]) {
        dfs(a, v, adyList);
    }
    v.push_back(nnode);
}

void Kosaraju(int n) {
    visited = vi(n, 0);
    stack<int> s = stack<int>();
    sccs = vector<vi>();

    vector<int> postorder;
    for (int i = 0; i < n; ++i) {
        dfs(i, postorder, adyList);
    }
    reverse(all(postorder));

    vector<vi> rAdyList = vector<vi>(n, vi());
    for (int i = 0; i < n; ++i) {
        for (auto v : adyList[i]) {
            rAdyList[v].push_back(i);
        }
    }

    visited = vi(n, 0);
    vi data;
    for (auto a : postorder) {
        if (!visited[a]) {
            data = vi();
            dfs(a, data, rAdyList);
            if (!data.empty())
                sccs.pb(data);
        }
    }
}
```

Running time: $\mathcal{O}(V + E)$

(V = vertices, E = edges)

Remarks

- The SCCs of any graph form a DAG

2.6 Kruskal's

Kruskal's algorithm finds a minimum spanning forest of the given graph. To do so, it uses UFDS to keep track of which nodes are already connected.

2.6.1 UFDS

UFDS (Union-find data structure) is a data structure that stores a partition of the vertices into sets such that all vertices in the same set are connected. Each node u will have two properties:

- *Parent*. It is the representative of the partition that contains u.
- *Rank*. It is an upper bound of the height of the "tree" that starts on u.

```
20 const int lim = 200000;
21 int parent[lim];
22 int rankk[lim];
23
24 void initialize(int n) {
25     for (int i = 0; i < n; ++i) {
26         rankk[i] = 0;
27         parent[i] = i;
28     }
29 }
30 int find(int x) {
31     if (parent[x] == x)
32         return x;
33     else
34         return parent[x] = find(parent[x]);
35 }
36
37 void Union(int a, int b) {
38     int pa = find(a);
39     int pb = find(b);
40     if (pa == pb) {
41         return;
42     }
43     if (rankk[pa] > rankk[pb]) {
44         parent[pb] = pa;
45     } else if (rankk[pa] < rankk[pb]) {
46         parent[pa] = pb;
47     } else {
48         parent[pa] = pb;
49         rankk[pb]++;
50     }
51 }
```

Running time: $\mathcal{O}(E \log V)$
(V = vertices, E = edges)

Explanation

There are two main optimizations that are applied in this implementation:

- *Union by rank*. When we join two partitions, we have to choose a node to represent the new partition. In order to choose between the two parents, we use their rank.

  Our goal is to minimize the height of the trees that start at every node. Therefore, we pick the node with the highest rank as the parent. This choice ensures that the rank of both parents will not increase.

  However, if both parents have the same rank, we can choose either of them.

- *Path shortening*. The `find` function updates the value of the parent of each node so that it does not point to it's "immediate" parent but to the highest possible ancestor. This difference increases the efficiency of subsequent executions of the `find` routine.

2.6.2 Kruskal's

Using UFDS, the implementation of Kruskal is trivial. We just edges that connect vertices that are not connected already until all vertices are connected.

```
20 ll kruskal(vector<pair<pii, int>> &edgeList) {
21     int cost = 0;
22
23     auto cmp = [](pair<pii, int> a, pair<pii, int> b) { return a.se < b.se; };
24     sort(all(edgeList), cmp);
25
26     for (auto a : edgeList) {
27         if (find(a.fi.fi) != find(a.fi.se)) {
28             //Not connected already -> connect them
29             Union(a.fi.fi, a.fi.se);
30             cost += a.se;
31         }
32     }
33     return cost;
34 }
```

Running time: $\mathcal{O}(\max\{E, V\} \log(\max\{E, V\}))$
(V = vertices, E = edges)

2.7 Flows and cuts

In this section we will consider graphs as networks of pipes and the "weights" of the edges will be their capacity.

2.7.1 Edmonds-Karp's (Max Flow)

The first problem that we have to consider is how to find the maximum flow from one source node to a sink node. Edmonds-Karp's is an implementation of Ford-Fulkerson's that solves this problem.

This algorithm has some major caveats: it requires the use of an adjacency matrix and it is not as fast as other options.

To improve its running time, this implementation uses both an adjacency matrix and adjacency list. However, we have to be careful when creating the adjacency list since for every forward edge, we need a backward edge that will start with a capacity of 0 but its capacity may increase.

Therefore, the code for adding edges would be the following:

```
// Add a directed edge
void addedgeUni(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjMat[orig][dest] = flow;
    adjList[dest].pb(orig);   //Add edge for residual flow
}
// Add a bidirectional edge
void addEdgeBi(int orig, int dest, ll flow) {
    adjList[orig].pb(dest);
    adjList[dest].pb(orig);
    adjMat[orig][dest] = flow;
    adjMat[dest][orig] = flow;
}
```

```cpp
vector<vector<ll>> adjList;
vector<vector<ll>> adjMat;

void initialize(int n) {
    adjList = decltype(adjList)(n);
    adjMat = decltype(adjMat)(n, vector<ll>(n, 0));
}

map<int, int> p;
bool bfs(int source, int sink) {
    queue<int> q;
    vi visited(adjList.size(), 0);
    q.push(source);
    visited[source] = 1;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (u == sink)
            return true;
        for (auto v : adjList[u]) {
            if (adjMat[u][v] > 0 && !visited[v]) {
                visited[v] = true;
                q.push(v);
                p[v] = u;
            }
        }
    }
    return false;
}
int max_flow(int source, int sink) {
    ll max_flow = 0;
    while (bfs(source, sink)) {
        ll flow = inf;
        for (int v = sink; v != source; v = p[v]) {
            flow = min(flow, adjMat[p[v]][v]);
        }
        for (int v = sink; v != source; v = p[v]) {
            adjMat[p[v]][v] -= flow; // Decrease capacity forward edge
            adjMat[v][p[v]] += flow; // Increase capacity backward edge
        }
        max_flow += flow;
    }
    return max_flow;
}
```

Running time: $\mathcal{O}(VE^2)$

(V = vertices, E = edges)

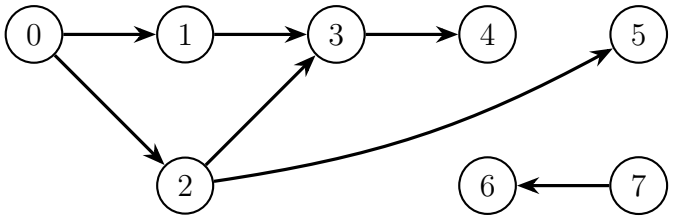
Remarks

- If we have multiple sources $s_1, \ldots, s_n$ and multiple sinks $t_1, \ldots t_n$, we can use the same algorithm. We just have to create a source s that connects to s_i with edges of infinite capacity and a sink t such that s_i is connected to t with edges of infinite capacity.
- If we have vertex capacities, we can split the vertex v into two vertices: v_{in} and v_{out}, connected with an edge that has the vertex capacity as a capacity. Then we connect the incoming edges to v_{in} and the out-coming edges to v_{out}

2.8 Directed Acyclic Graphs (DAG)

2.8.1 Topological sort

A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge u $\rightarrow$ v, u comes before v in the ordering. A topological sorting is possible if and only if the graph has no directed cycles, that is, it has to be a DAG. As an example:



0	1	2	3	4	5	6	7
0	1	2	3	4	5	7	6

Regarding the algorithm to obtain the topological sorting, there are two alternatives:

- A modified version of DFS
- Kahn's algorithm

Using DFS

The key observation is:

> *When DFS closes a node, all nodes that are accessed from that node are already closed*

Therefore if we order the vertices regarding the DFS closing time, we can be sure that if there exists an edge u $\rightarrow$ v, v will be placed before u since it must have been closed before v was closed. Now we should realize that to get the topological sorting, we only need to reverse this ordering.

```
vi topoSort;
vi visited;
vector<vi> adyList;
void dfs(int node) {
    if (visited[node]) {
        return;
    }
    visited[node] = true;
    for (auto a : adyList[node]) {
        dfs(a);
    }
    topoSort.push_back(node);
}
void topologicalSort() {
    int n = adyList.size();

    visited = vi(n, 0);
    topoSort.clear();

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(all(topoSort));
}
```

Running time: $\mathcal{O}(E + V)$
(V = vertices, E = edges)

Kahn's algorithm

Kahn's algorithm is a very simple algorithm. It simply keeps track of the in-degree of every vertex and only adds to the topological sort list vertices that do not have any incident edges.

Furthermore, this algorithm can be easily adapted to store all possible topological orderings by using a recursive function and running through all possibilities at each step

```
vi topoSort;
vi visited;
vector<vi> adyList;
void dfs(int node) {
    if (visited[node]) {
        return;
    }
    visited[node] = true;
    for (auto a : adyList[node]) {
        dfs(a);
    }
    topoSort.push_back(node);
}
void topologicalSort() {
    int n = adyList.size();

    visited = vi(n, 0);
    topoSort.clear();

    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(i);
        }
    }
    reverse(all(topoSort));
}
```

Running time: $\mathcal{O}(E + V)$
(V = vertices, E = edges)

2.9 Bipartite Graphs

A graph is bipartite if its vertices can be split into two different so that no two nodes in the same set are connected by an edge.

Definitions

It is important to note that in this section a path (unless specified otherwise) will be a simple path (not containing repeated vertices or edges).

- *Matching*. Set of pairwise non-adjacent edges in a graph. That is to say, no more than one edge from the set should be incident to any vertex of the graph.
- *Saturated* (a vertex with respect to some matching). A saturated vertex is a vertex that is an endpoint of an edge contained in the matching.
- *Alternating path* (with respect to some matching). Path in which the edges alternately belong / do not belong to the matching.
- *Augmenting path* (with respect to some matching M). Alternating path whose initial and final vertices are unsaturated (they do not belong to the matching). That is to say, if we represent the edges already included in M as I and the new ones as U, it follows this pattern:

  $$U - I - U - \cdots - U - I - U$$

  We can also notice that there will be $\left\lfloor \frac{|P|}{2} \right\rfloor$ edges already included and $\left\lceil \frac{|P|}{2} \right\rceil$ new edges.
- *Independent set*. Set of vertices in which no two vertices represent an edge in the original graph.
- *Vertex coverage*. Set of vertices C that fulfills the following property: Every vertex of G is incident on, at least, one vertex of C.

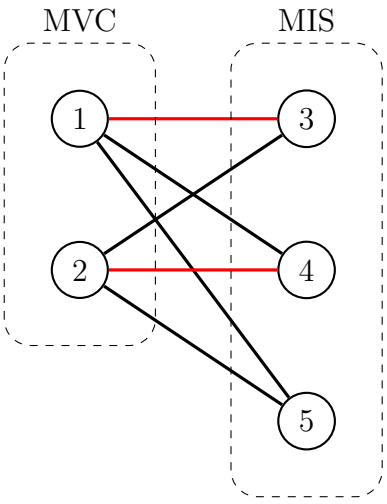
From these definitions, we can derive the following features in a bipartite graph:

- *Maximum cardinality bipartite matching (MCBM)*. Matching that contains the maximum number of edges.
- *Maximum independent set (MIS)*. Independent set that fulfills that if any other vertex was to be added to it, there would be an edge between two of the vertices of the set.
- *Minimum vertex coverage (MVC)*. Vertex coverage with the minimum number of vertices.

Furthermore, it can be proved that even though these "sets" may not be unique, their cardinality is. Additionally, they are closely related among them:

$$|\text{MIS}| = |\text{V}| - |\text{MCBM}| \qquad |\text{MCBM}| = |\text{MVC}|$$

Example



In this example we have a bipartite graph where nodes 1 and 2 belong to the first set of vertices and nodes 3, 4 and 5 to the second set.

- Since there are only two vertices in the first set, the MCBM has to be at most two. We have found a matching of size two (in red), so that must be the solution.
- The MIS has to be one of the two sets of vertices since all vertices from one set are connected to all vertices from the second set. We pick the maximum one.
- Finally, the MVC has to be the smallest set for the same reasons.

Finally, we can also notice how the MCBM is not unique even though its cardinality is unique.

Berge's lemma

A matching M is maximum $\iff$ there is no augmenting path relative to the matching M

Proof

We can prove it by contradiction:

$\implies$ Let there be an augmenting path P relative to M. This path will necessarily be of odd length (it has one more edge not in M than in M). We can now create a new matching $M' = M \oplus P = (M \setminus P) \cup (P \setminus M)$.

Now we have to see that this matching is valid. In order for it not to be valid, there must be a vertex such that two edges in M' are adjacent to. Furthermore, one of those edges must belong to $P \setminus M$ while the other one belongs to M. However, the second one must then also belong to P (P alternates edges) and will not be in $M \setminus P$. Therefore, it cannot be in M' and we reached a contradiction

Finally, we can see that there are $|M| + 1$ edges in M' since:

- By subtracting P from M we delete $\left\lfloor \frac{|P|}{2} \right\rfloor$ edges
- By adding $P \setminus M$, we add $\left\lceil \frac{|P|}{2} \right\rceil$ edges

Therefore, M could not be a maximum matching

$\Longleftarrow$ Let there be a matching M' of greater cardinality than M. We can consider the subgraph $Q = M \oplus M'$. Any vertex in this subgraph has a maximum degree of 2, which means that all connected components in it are either:

- An isolated vertex
- a (simple)path whose edges are alternate from M and M'
- A cycle of even length whose edges are alternate from M and M'

However, since there are more edges from M' than from M, there must be at least one component with more edges from M' than from M. Since such path must be alternating, it will have initial and final vertices unsaturated by M, making it an augmenting path for M.

□

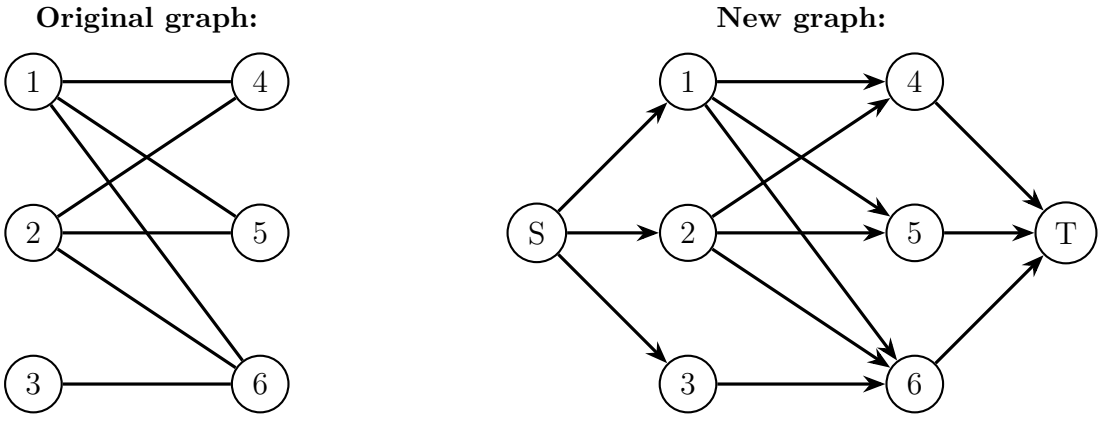
2.9.1 Algorithms to find MCBM

Reduce to Max-Flow

The bipartite matching problem can be easily reduced to a max-flow problem. We only have to:

- Make the edges between the two components directed from the first component to the second (all edges must go in the same direction)
- Connect the source to all nodes in the first component.
- Connect the nodes in the second component to the sink.
- Set the capacities of all edges to 1.

For instance:



After this process, we can run the normal max-flow algorithm and the maximum flow between the source and the sink will be the result. This will have a time complexity of $\boldsymbol{\mathcal{O}(VE^2)}$ using Edmonds-Karp's or $\boldsymbol{\mathcal{O}(V^2E)}$ using Dinic's

Kuhn's Algorithm

On a high level, the algorithm implements the following pseudocode (where M is the matching):

```
M ← ∅
while (able to find augmenting path P){
        M ← P ⊕ M
}
```

This is a direct application of Berge's lemma; however, if we were to implement it naïvely, we could end up with a time complexity of $\mathcal{O}(V^2E)$ since there will be at most V augmenting paths and to find each of them, we would need to check for paths that start on each node and then run DFS to find the actual path. To make it faster, it can be implemented using the following pseudocode (where n_1 is the size of the first set in the graph):

```
M ← ∅
for (i = 0 ... n1){
        P ← augmenting path that starts on i
        if(P ≠ ∅){
                M ← P ⊕ M
        }
}
```

To understand why this implementation should work, it is necessarily to make a few observation:

- The augmenting path found in each iteration starts on a vertex in the first set, then traverses some number of vertices that are already saturated and it ends on a vertex of the second set that is not saturated.
- When we update M with the new P, no vertices that were saturated before will be unsaturated now. Since all saturated vertices in P are connected by two adjacent edges, one of them will still be present in P $\oplus$ M since it could not be present on M before.
- If there is no augmenting path that starts from i in the current iteration, it will be impossible to find one in a some future iteration. This is because no vertex can go from being saturated to being unsaturated in this process.

With all of this in mind, we can obtain a c++ implementation that runs on $\mathcal{O}(EV)$. Furthermore, if we make sure that the first set of the bipartite graph is the one with less elements, the runtime will be faster since the iterations of the outer loop will decrease.

As we can see, the search for an augmenting path from a node will try all options to pair that node to a node in the second set. For each one of them, if they are already matched to some node in the first set, it will try to create an augmenting path from there. The algorithm will stop when it finds an unsaturated vertex or runs out of vertex to check.

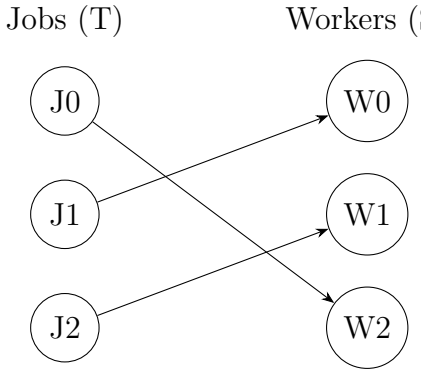
```cpp
// List of vertices in the 2nd component connected to
// the i-th vertex in the first component
vector<vi> adyList;

// Index of the vertex in 1st component connected to the
// i-th vertex in the second component
vector<int> match;
vector<bool> used;
bool augmentPath(int current) {
    if (used[current]) {
        return false;
    }
    used[current] = 1;
    for (int to : adyList[current]) {
        if (match[to] == -1 || augmentPath(match[to])) {
            match[to] = current;
            return true;
        }
    }
    return false;
}
int calculateMCBM(int n1, int n2) {
    match = vi(n2, -1);
    int res = 0;
    for (int i = 0; i < n1; ++i) {
        used.assign(n1, false);
        res += augmentPath(i);
    }
    return res;
}
```

Running time: $\mathcal{O}(EV)$
(V = vertices, E = edges)

2.9.2 Assignment problem

There are n workers that should complete n jobs. Each assignment (worker, job) has a given cost and each worker can only perform one job (and vice versa). The goal is to minimize the total cost of performing all jobs.

This problem can be modeled as a bipartite graph where the jobs and workers are represented by vertices. In such graph, we can encode the assignment of a job to a worker by adding an edge between those two vertices. For example:



Costs			
	W0	W1	W2
J0	1	3	5
J1	1	10	20
J2	10	3	10

Definitions

- A complete matching M is a set of edges such that all jobs are paired with exactly one worker.
- A perfect matching M is a complete matching such that the added cost of all the edges is maximum.
- The set of all nodes that represent jobs is T.
- The set of all nodes that represent workers is S.
- The cost of adding an edge between two vertices v, w ($v \in S \wedge w \in T$ or $v \in T \wedge w \in S$) is represented as $c(v, w)$.
- A function $y : (S \cup T) \to \mathbb{R}$ is a potential if $y(v) + y(w) \leq c(v, w)$
- The value of a potential y is: $V_y = \sum_{v \in S \cup T} y(v)$
- An edge $v - w$ is tight with respect to a potential y if $c(v, w) = y(v) + y(w)$

Assumptions

- All costs in the cost matrix are positive. This is not required in the final implementation but simplifies the proof. Moreover, it is trivial to make the costs positive by adding a large positive value to all of them.
- There is the same number of both workers and jobs. This simplifies the proof but is not required. If we have less jobs than workers, we can always add more jobs with cost 0 to all workers.

Analysis of the algorithm

The goal of the algorithm is to find a perfect matching. In order to do so, we can observe that:

The cost C of a perfect matching M is greater or equal to the value of any potential y

We can easily prove this statement. Since M pairs each job with one worker, we can enumerate identify the worker assigned to the job v as $w(v)$:

$$V_y = \sum_{u \in S} y(u) + \sum_{v \in T} y(v) = \sum_{v \in T} y(v) + y(w(v)) \leq \sum_{v \in T} c(v, w(v)) = C$$

From that claim, we can deduce that if the cost of a complete matching M equals the value of a potential, M is perfect Furthermore, if we analyze the previous inequality, we can observe that in a perfect matching, there must be a potential y such that all edges belonging to the matching M are tight with respect to y. Thus:

A complete matching M formed by tight edges with respect to some potential y is a prefect matching

Using this insight we can code a constructive algorithm that produces complete matchings that only consist of tight edges. On a high level, this algorithm performs the following operations:

```
for each job i in [0,n):
        run O(n^2) Dijkstra to calculate the "closest"
        not matched worker

        update the current matching and potential to obtain a
        matching that matches all [0, i] jobs using tight edges
```

The implementation of this algorithm required the following arrays:

- `matchedJob[i]`: Job matched to the i-th worker
- `jobPotential[i]`: Value of the potential evaluated in the i-th job. We will see in the implementation that this is always a negative value (it is only decreased during the algorithm)
- `workerPotential[i]`: Value of the potential evaluated in the i-th worker. We will see in the implementation that this is always a positive value (it is only increased during the algorithm)
- For each iteration of the outer loop:
  - `dist[i]`: Minimum distance from the visited
  - `from[i]`: Worker from which we found the current distance to the i-th worker.
  - `seen[i]`: Whether the i-th worker has been visited

Pseudocode of the implementation

```
int hungarian(costMatrix):
        matchedJob[*] ← -1
        jobPotential[*] ← 0
        workerPotential[*] ← 0

        for maxJob in [0,n):  # outer loop
                currentWorker = helperNode
                matchedJob[helperNode] = maxJob
                dist[*] ← ∞
                from[*] ← -1
                seen[*] ← 0

                while (currentWorker not matched):  # Dijkstra loop
                        seen[currentWorker] = 1
                        currentJob = matchedJob[currentWorker]
                        delta = ∞

                        for worker in [0,n):
                                if seen[worker]: continue
                                candidateDistance = costMatrix[currentJob][worker]
                                candidateDistance -= jobPotential[currentJob]
                                candidateDistance -= workerPotential[worker]
                                if candidateDistance < dist[worker]:
                                        dist[worker] = candidateDistance
                                        from[worker] = currentWorker
                                if dist[worker] < delta:
                                        delta = dist[worker]
                                        nextWorker = worker

                        for worker in [0, n):
                                if seen[worker]:
                                        jobPotential[matchedJob[worker]] += delta
                                        workerPotential[worker] -= delta
                                else:
                                        dist[worker] -= delta

                        currentWorker = nextWorker

                while currentWorker ≠ helperNode:
                        worker = from[currentWorker]
                        matched[currentWorker] = matched[worker]
                        currentWorker = worker

        return workerPotential[helperNode]
```

Outer loop

The outer loop has two invariants that we have to maintain. At the beginning of the k-th iteration, the following properties must hold true:

1. We have a matching consisting of k tight edges that is complete with respect to the first $k-1$ jobs.
2. The potential function given by arrays `jobPotential` and `workerPotential` fulfills the potential property.

In each iteration of the outer loop, we have to alter the already formed matching to add exactly one more job and one more worker. The job that we have to add is given by the `maxJob` loop counter.

However, deciding which worker to add is more challenging since the optimal solution might involve changing some of the previous assignment. This is the problem that this modified Dijkstra aims to solve.

The cost function for Dijkstra's

During the search process, the cost of an edge that goes from v to w will be considered to be $c(v,w) - y(v) - y(w)$. This is the increment in the cost of the matching if we add that edge to it. If we analyze it case by case:

- When there are no edges in the matching that touch either vertex, the potencial of the vertices will be 0 and the cost will not be altered.
- When the job was already touched by an edge, the cost of this edge must be decreased since if we add this edge, we will eliminate the previous paring for the job, which already had a cost. This is the effect of subtracting the potential of the job (which is a positive value).
- When the worker was already picked, the cost of this edge must be increased since adding this edge will imply deleting the previous pairing and adding an additional one that pairs the job that no longer is paired with another worker. Since the potential value of a worker is negative, subtracting it will increase the cost of the edge.

Using this cost definition enables the algorithm to search for the minimal increase to the cost of the matching.

Modified $\mathcal{O}(n^2)$ Dijkstra's

To start the Dijkstra, we use a helper node that will be matched to the new job. This helper node has two main uses: it simplifies the code of the algorithm by making the first iteration more similar to the following ones and it's potential value stores the cost of the entire matching.

Now the algorithm can start. In each iteration, the algorithm will have a `currentJob` and a `currentWorker` value, where the `currentJob` value is the job matched to the current worker. It should be noted that for all intensive purposes, either node can be considered the current node since they are joint by an edge of "cost" 0.

After the innermost for loop, the `dist` array will be updated to contain the lowest cost from the explored jobs to each worker. Additionally, the `from` array might also change to reflect the changes in the distance vector. Finally, the `nextWorker` and the `delta` variables will be updated to the "closest" worker and the cost of reaching that worker.

Now we have to update the potential values. Since it takes at least `delta` to reach any other worker, we can increase the potential of all visited jobs by `delta`. However, all those nodes are connected to a worker with a tight edge since they are already paired. Therefore, in order to maintain the potential property, we have to decrease the potential at the other end of those tight edges.

Finally, since we are updating the potentials, we also have to change the distances accordingly. Since the potentials of all visited jobs have been increased by `delta`, the minimum distances to all workers are decreased by the same amount. This update is the price we have to pay to be able to update the "cost" of the edges of the graph in the middle of the search.

To finish the iteration we have to check if the current worker is already matched to some job. If it is, we move to the worker and conduct at least another iteration. Otherwise, we can stop the algorithm since we have obtained the worker that can be matched with a job while increasing the cost of the matching by the minimum amount.

If we look back at the invariants of the outer loop, we can see that the second invariant already holds true after this process. However, to make the first one true, we still have to update the `matchedJob` array. In order to do so, we can simply backtrack from the last worker using the `from` array.

C++ implementation

This implementation follows the previous pseudocode but there are some aspects that should be kept in mind:

- The input matrix must have the following format:

  `matrix[i][j]` $\equiv$ Cost of having the i-th job done by the j-th worker

- If the objective is the maximum cost, the entries of the cost matrix should be multiplied by -1.
- The algorithm can handle cases where there are more workers than jobs to get done

```cpp
// The rows are jobs, the columns are workers
pair<ll, vl> hungarian(vector<vl> &matrix) {
    int n = matrix.size(), m = matrix[0].size();
    vl jobP(n), workerP(m + 1), matched(m + 1, -1);

    vl dist(m + 1, inf);
    vi from(m + 1, -1), seen(m + 1, 0);

    for (int i = 0; i < n; ++i) {
        int cWorker = m;
        matched[cWorker] = i;
        std::fill(all(dist), inf);
        std::fill(all(from), -1);
        std::fill(all(seen), false);

        while (matched[cWorker] != -1) {
            seen[cWorker] = true;
            int i0 = matched[cWorker], nextWorker = -1;
            ll delta = inf;

            for (int worker = 0; worker < m; ++worker) {
                if (seen[worker])
                    continue;
                ll candidateDistance = matrix[i0][worker];
                candidateDistance += -jobP[i0] - workerP[worker];

                if (candidateDistance < dist[worker]) {
                    dist[worker] = candidateDistance;
                    from[worker] = cWorker;
                }
                if (dist[worker] < delta) {
                    delta = dist[worker];
                    nextWorker = worker;
                }
            }
            for (int j = 0; j <= m; ++j) {
                if (seen[j]) {
                    jobP[matched[j]] += delta;
                    workerP[j] -= delta;
                } else {
                    dist[j] -= delta;
                }
            }
            cWorker = nextWorker;
        }
        while (cWorker != m) {
            int prevWorker = from[cWorker];
```

```
            matched[cWorker] = matched[prevWorker];
            cWorker = prevWorker;
        }
    }
    ll ans = -workerP[m];
    vl rowMatchesWith(n);
    for (int j = 0; j < m; ++j) {
        if (matched[j] != -1) {
            rowMatchesWith[matched[j]] = j;
        }
    }
    return {ans, std::move(rowMatchesWith)};
}
```

Remarks

- Sometimes, it might be useful to add workers to represent actions that may occur sequentially.

Chapter 3

Mathematics

3.1 General calculations

3.1.1 Binary exponentiation

Binary exponentiation calculates a power in logarithmic time. Furthermore, it can be used for modular arithmetic:

```
20 ll power(ll base, ll exp) {
21     if (exp == 0) {
22         return 1;
23     }
24     ll res = power(base, exp / 2);
25     res = (res * res) % mod;
26     if (exp % 2) {
27         res *= base;
28     }
29     return res % mod;
30 }
```

Running time: $\mathcal{O}(\log(\text{exp}))$

3.2 Modular arithmetic

3.2.1 Inverses

To calculate a modular inverse, we will use Fermat's little theorem:

$$a^{p-1} \equiv 1 \bmod p \implies a^{p-2} \equiv a^{-1} \mod p$$

If we combine this fact with binary exponentiation, we can obtain the modular inverse in logarithmic time:

```
ll inverse(ll num) {
    return power(num, mod - 2);
}
```

Running time: $\mathcal{O}$(log(mod))

3.3 Catalan numbers

We define the nth Catalan number as:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{n+1}\frac{(2n)!}{n!\,n!}$$

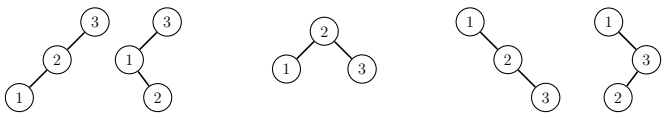
We can also define them recursively:

$$C_0 = 1 \qquad C_{n+1} = \sum_{i=0}^{n} \left(C_i\, C_{n-i}\right)$$

They can be used to solve many different problems. For instance:

- Number of different binary trees of n nodes. We can look at the specific case $n = 3$. As we can see in the figure below, $C_3 = 5$. Furthermore, we can clearly identify the recursive relationship:

$$C_3 = (3 \text{ is root}) + (2 \text{ is root}) + (1 \text{ is root}) = C_2 \cdot C_0 + C_1 \cdot C_1 + C_0 \cdot C_2$$



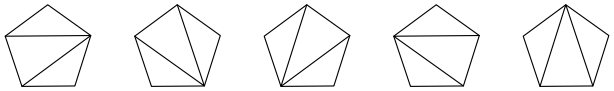
- Number of expressions that contain n pairs of parenthesis correctly paired. For instance, for $n = 3$, there only $C_3 = 5$ valid arrangements are:

$$()()() \quad ()(()) \quad (())() \quad ((())) \quad (()())$$

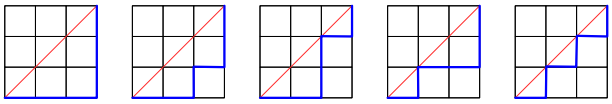
- Number of different ways of arranging $n+1$ factors in parenthesis In the case of $n = 3$, we can use the factors a, b, c, d. Then, we will have the following options:

$$(ab)(cd) \quad a(b(cd)) \quad ((ab)c)d \quad (a(bc))d \quad a((bc)d)$$

- Number of ways to split a convex polygon of $n + 2$ sides into triangles.



- Number of paths from the bottom-left corner to the top-right corner of a $n \times n$ grid moving only up or right at every point and while not reaching the diagonal.



3.4 Permutations

3.4.1 Converting one permutation into another

In this problem we are given two permutations of the same size and we are tasked with converting one of them into the other. In order to do so, we will swap the elements at two positions in the permutation zero or more times.

	0	1	2	3	4	5	6
A =	1	2	3	4	5	7	6

	0	1	2	3	4	5	6
B =	1	3	2	4	5	7	6

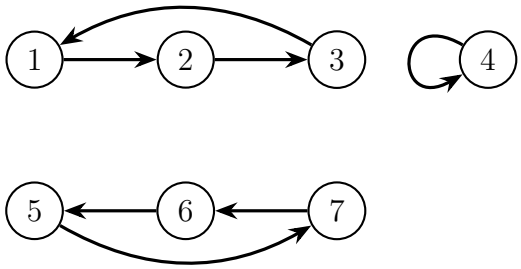
In this case, the most efficient way is to swap positions 1 and 2 of permutation B to get permutation A. From examples like this one, we can deduce the following observations:

- One swap will fix at most two numbers out of place.
- We will need at most $n-1$ swaps to change one permutation into the other. Note that we can "skip" the swap that corresponds to the last element since, at that point, by the Pigeonhole Principle, all elements must be in the correct position.

To solve the problem, we shall change the representation to a graph where the edge `a` $\rightarrow$ `b` represents that there exists a position `j` such that `A[j]=a, B[j]=b`. Since the previous example was extraordinarily simple, we will present a new one to show this process:

	0	1	2	3	4	5	6
A =	1	2	3	4	5	6	7

	0	1	2	3	4	5	6
B =	2	3	1	4	7	5	6



This graph will fulfill the following properties:

- All edges are part of a cycle (possibly of length 1). This is due to the fact that all vertices have an in-degree and out-degree of 1.
- Since all edges form cycles, connectivity and strong connectivity are equivalent. In particular, we can make the edges undirected without altering the connectivity.

The final piece of the puzzle is noticing that the elements that form a cycle of length k need exactly $k-1$ swaps to be rearranged. In order to prove this, we can use induction.

If $k \leq 2$, it is trivial. We can now assume that we have a cycle of length k. When we swap the first two elements, we can either fix one element that was out of place or two. However, if we had fixed two, that would mean that those two were interchanged, in which

case they would form a 2-cycle and they could not be part of any larger cycle. Therefore, we can only fix one element in this first swap. After this swap, the cycle has length $k-1$ and, by the inductive hypothesis, we will need $(k-1)-1$ moves to rearrange all the elements. Thus, if we add the swap that we already performed, we obtain the expected result of $k-1$ swaps.

To get the final formula, we observe that if there is a big cycle that includes all elements, we would need $n-1$ swaps and every cycle that we introduce decreases the number of swaps required by one. Therefore:

$$\#\text{ Swaps} = n - \#\text{ Components}$$

To implement this formula we can construct the graph using undirected edges and count the components applying UFDS

```
55 int swapsPermutation(vi perm1, vi perm2) {
56     int n = perm1.size();
57     UFDS::initialize(n + 1);
58     int components = n;
59     for (int i = 0; i < n; ++i) {
60         if (UFDS::find(perm1[i]) != UFDS::find(perm2[i])) {
61             components--;
62             UFDS::Union(perm1[i], perm2[i]);
63         }
64     }
65     return n - components;
66 }
```

Running time: $\mathcal{O}(n)$

3.4.2 Converting one permutation into another using adjacent swaps

This is a very similar problem; however, now we can only swap elements that are next to each other in the array. Furthermore, in this analysis, we will reduce the problem to calculating the minimum number of adjacent swaps required to sort a permutation. This is equivalent since we can use a bijective application to "redefine" the order according to how the elements are present in the first permutation.

The key observation in this case is the fact that the solution is the number of inversions in the permutation. We can define an inversion as a pair of indices (i, j) such that $i < j$ but $A_i > A_j$. We can easily notice that (choosing optimally), each adjacent swap will fix exactly one inversion since it will fix the relative order between two elements.

Furthermore, this idea is surprisingly easy to implement since we can use a modified version of Merge Sort that counts the swaps that would have been required to place an element in a specific position while maintaining a log-linear complexity.

To do so, we can just count at each step how many position we had to move the elements of the left half of the array to sort them.

```cpp
ll mergeSort(int left, int right, vi &elems) {
    if (left == right) {
        return 0;
    }
    int mid = (left + right) / 2;
    ll res = 0;
    res += mergeSort(left, mid, elems);
    res += mergeSort(mid + 1, right, elems);

    int index = 0;
    vi aux(right - left + 1, 0);
    int i1 = left, i2 = mid + 1;
    while (i1 != mid + 1 || i2 != right + 1) {
        int *picked;
        if (i1 > mid) {
            picked = &i2;
        } else if (i2 > right) {
            picked = &i1;
        } else if (elems[i1] <= elems[i2]) {
            picked = &i1;
        } else {
            picked = &i2;
            res += abs((*picked) - (left + index));
        }
        aux[index++] = elems[*picked];
        (*picked)++;
    }
    for (int i = 0; i < right - left + 1; ++i) {
        elems[i + left] = aux[i];
    }
    return res;
}

ll countAdjacentSwaps(vi &elems) {
    return mergeSort(0, int(elems.size()) - 1, elems);
}
```

Running time: $\mathcal{O}(n \log(n))$

Chapter 4

Strings

4.1 Definitions

- A proper prefix of a string is a prefix that is not equal to the string itself.

4.2 Prefix function

Given a string s, we can calculate its prefix function π for every index in the string. For a given index i, $\pi(i)$ is the length of the longest proper prefix of s that is also a suffix of $s[0 \dots i]$. Formally:

$$\pi(i) = \max_{k=0 \dots i} \left\{ k : s[0 \dots k-1] = s[i-(k-1) \dots i] \right\} \quad i \in \left[0, |s|\right)$$

For example:

$i = 0$ a a b a a a b $\pi(0) = 0$

$i = 1$ a a b a a a b $\pi(1) = 1$

$i = 2$ a a b a a a b $\pi(2) = 0$

$i = 3$ a a b a a a b $\pi(3) = 1$

$i = 4$ a a b a a a b $\pi(4) = 2$

$i = 5$ a a b a a a b $\pi(5) = 2$

$i = 6$ a a b a a a b $\pi(6) = 3$

4.2.1 Remarks

- When going from one index to the next, the prefix function can never increase by more than one.
- It can be used to quickly calculate all the prefixes that are also suffixes of $s[0 \dots i]$, not only the longest one.

  Let $k = \pi(i)$. Then $s[0 \dots k]$ is a suffix of $s[0 \dots i]$ and all other prefixes that are also suffixes of $s[0 \dots i]$ must have a strictly shorter length. Therefore, they have to be also suffixes of $s[0 \dots k]$

Thus, the prefixes that are also suffixes, in descending order of length are:

$$s[0 \dots \pi(i)-1] \quad s[0 \dots \pi(\pi(i)-1)-1] \quad s[0 \dots \pi(\pi(\pi(i)-1)-1)-1] \quad \dots$$

4.2.2 Calculation

To calculate the prefix function of an index we have t:o

1. Check if we can extend the prefix of the previous index, which would lead to an optimal solution. To check if we can do this, we only have to check the following condition: `s[i] == s[π(i-1)]`
2. Otherwise, we will check if it is possible to extend one of the smaller suffixes of the previous index. To do so, we can iterate and use the procedure described in the second remark to check all the suffixes.

The following implementation will work with any container that provides a [] operator.

```
20 template <typename T> vi prefixFun(const T &s, int n) {
21     vi res(n);
22     for (int i = 1; i < n; ++i) {
23         int j = res[i - 1];
24         while (j > 0 && s[i] != s[j]) {
25             j = res[j - 1];
26         }
27         res[i] = j + (s[i] == s[j]);
28     }
29     return res;
30 }
```

Running time: $\mathcal{O}(n)$

It is important to note that $j-1$ would be the index where the suffixes that we consider in the `while` loop end. However, changing the variable in this way simplifies the code. If j were to encode the ending of the suffix that is being tested to be extended, the code would be:

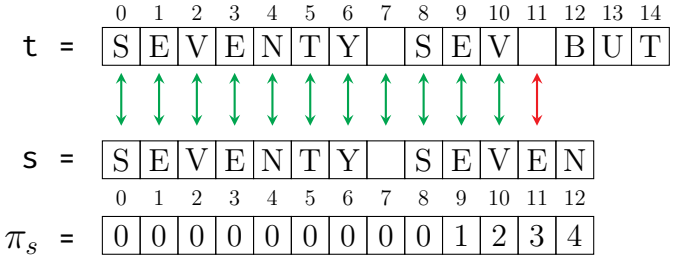
```
1         int j = res[i-1]-1;
2         while (j >= 0 && s[i] !=s[j+1]){
3             j = res[j] - 1;
4         }
5         res[i] = j + (s[i] == s[j]);
```

4.2.3 KMP

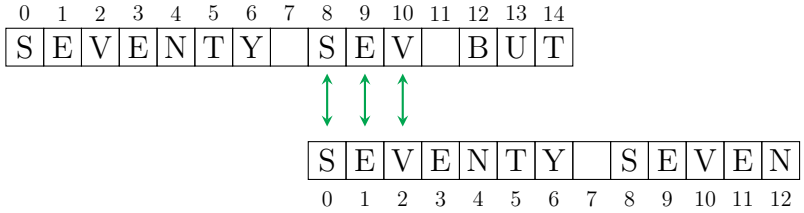
The Knuth-Morris-Pratt algorithm searches for a string s in a piece of text t. This algorithm has a worse average performance compared to the naïve string matching algorithm but its worst-case complexity is linear instead of quadratic.

In order to do so, the algorithm uses the prefix function to avoid repeating comparisons when a character has already been matched at some point. In particular, when we run into a character that does not match, the prefix function will tell us how many of the last characters that we have matched are a prefix, and when we move the position of s, we do not have to mark those characters as matching again.

For instance, we could start matching the following string s against the text t. Until we reach position 11 in the strings, all characters will match and KMP will behave exactly as the naïve string matching algorithm. However when we reach this position, KMP uses the



prefix function to search for the longest prefix that ends at the last matched character. In this case, since the prefix function at $j = 10$ is 3, we will shift s so that the next character of s that we match is 3.



In this case, when we check the index $i = 11$ of t, we will see that there is no match, therefore, we would repeat the process and move s further to the right according to the prefix function until it matches of we reach the left end of s.

Implementation

This version of the algorithm counts how many times the pattern is present in the text. To execute it, we have to first calculate the prefix function and then run the following code:

```
template <typename T>
int kmpSearch(const T &text, int n, const T &pattern, int m,
              const vi &patternPre) {

    int count = 0;
    int j = 0;
    for (int i = 0; i < n; ++i) {
        while (j > 0 && text[i] != pattern[j]) {
            j = max(0, patternPre[j] - 1);
        }
        j += (text[i] == pattern[j]);
        if (j == m) {
            count++;
            j = patternPre[j - 1];
        }
    }
    return count;
}
```

Running time: $\mathcal{O}(|s| + |t|)$

4.2.4 Other applications

- Instead of using KMP, it is possible to concatenate $s + \# + t$ and calculate the prefix function of this new string. Where the string function is $|s|$, there is a match.
- *Number of occurrences of each prefix.* A prefix can appear as the prefix itself, as a maximum suffix of a substring or as a non-maximum suffix of a substring. The first two cases are rather simple. Regarding the last one, it is enough to observe that the inclusion relationship (as a suffix) is transitive. In this code ans[i] is the number of times that the prefix of length i occurs;

```
vl ans(n);
    for (int i = n - 1; i >= 0; --i) {
        ans[pref[i]]++; // Count the maximum suffix here
        if (i) {
            ans[pref[i - 1]] += ans[i];  // Add the times that this
        }                                // suffix appears to the smaller one
        ans[i]++; // Count the prefix itself
    }
```

Therefore, it can be solved in linear time

- *Number of different substrings in a string.* Add all characters one after another. At the i-th step, we have the string s_i. After adding a character c, $t_i = s_i + c$. Now, we reverse the string and calculate the prefix function. Its maximum value is the amount of newly added substrings that were already present. Therefore, the number of new substrings appearing at the i-th step is:

  $$|s| + 1 - \pi_{\max}$$

  After n steps, the solution is reached. Thus, this algorithm has a quadratic time complexity.

- *Compress a string.* Find a prefix t of the string s such that s can be formed by concatenating t several times.

  Compute the prefix function. Let $k = n - \pi(n-1)$. If $k \mid n$, k is the solution. Otherwise, there is no way to compress it.

4.3 Suffix Array

A suffix array is an array of integers that represent the starting indexes of all the suffixes of a given string. As an example, we can calculate the suffix array for `s = "abab"`.

Suffixes sorted by starting index

```
0  abaab
1  baab
2  aab
3  ab
4  b
```

Suffixes sorted lexicographically

```
2  aab
3  ab
0  abaab
4  b
1  baab
```

It clearly follows that the suffix array is `[2, 3, 0, 4 ,1]`.

4.3.1 Construction

To construct the suffix array of s we will not sort its suffixes but rather its cyclic shifts. However, to make sure that these operations are equivalent, we must append a character at the end of s that is strictly smaller than all other characters in s. It is common to use the symbol \$. After appending the \$, we can see how sorting the cyclic shifts of the string is equivalent to sorting the suffixes. The following figure illustrates this point with the same example as before.

Sorted by starting index

```
0  abaab$  abaab
1  baab$a  baab
2  aab$ab  aab
3  ab$aba  ab
4  b$abaa  b
```

Sorted lexicographically

```
2  aab$ab  aab
3  ab$aba  ab
0  abaab$  abaab
4  b$abaa  b
1  baab$a  baab
```

Now we can focus on sorting cyclic shifts. In order to sort the cyclic shifts, we will sort cyclic substrings of increasing length. Initially, we will sort the cyclic substrings of length 1. In the next iteration, we can use that information to sort cyclic substrings of length 2. This algorithm allows us to sort the cyclic shifts (cyclic substrings of length n) after $\lceil \log n \rceil + 1$ iterations.

A cyclic substring is perfectly defined by its starting index and its length. Note that since it is cyclic, when it reaches the end of the string, it will wrap to the beginning. Therefore, the cyclic substring of length 4 starting in index 3 of "abcde" would be "deab". We can denote $S(i, l)$ the cyclic substring of the string s that starts at index i with length l

In iteration k we want to sort the cyclic substrings of length 2^k in $\mathcal{O}(n)$ using the order of the cyclic substrings of length 2^{k-1}. In order to do so, we will take advantage of the fact that each substring that we are considering is comprised of two sub-substrings of length 2^{k-1}. In particular:

$$S(i, 2^k) = S(i, 2^{k-1}) + S(i + 2^{k-1} \bmod n, 2^{k-1})$$

Therefore, to compare two cyclic substrings of length 2^k we only need to compare the first half first and, if they are equal, compare the second half. In order to do so, we can associate each cyclic substring with a pair that contains the order of each of the two halves and sort by that pair.

To store the relative order between cyclic substrings in iteration k, two arrays are needed:

- **p**[]. This array stores a permutation of the cyclic substrings in which they would be sorted. Since there can be repeated elements, this array is not unique.
- **c**[]. This array stores the order of the cyclic substrings so that two equal substrings have the same value and a greater substring has a higher value. Formally, the constraints are:

  $$S(i, 2^k) < S(j, 2^k) \implies c[i] < c[j] \qquad S(i, 2^k) = S(j, 2^k) \implies c[i] = c[j]$$

  This array is necessary since equal substrings must be treated equally when the sorting in the next iteration is performed.

  We will refer to the value $c[i]$ as the class of $S(i, 2^k)$.

For instance, in the string `s = aabb`, we would have:

k	Cyclic substrings	p[*]	c[*]
0	[a, a, b, a]	[0, 1, 3, 2]	[0, 0, 1, 0]
1	[aa, ab, ba, aa]	[0, 3, 1, 2]	[0, 1, 2, 0]
2	[aaba, abaa, baaa, aaab]	[3, 0, 1, 2]	[1, 2, 3, 0]

The last remaining issue is how to sort the cyclic substrings. In the base case $k = 0$, each substring is a letter so we can simply use counting sort to sort them in linear time. However, in the following iterations, we have to sort pair of numbers in linear time. In order to do so, we will split the sorting process in two parts

1. Sort by the second element of the pair. Since the second elements were already sorted in the previous iteration, we just need to subtract 2^{k-1} from the indices in p. To see why this works, we can observe that all pairs have a different second element. Furthermore, we can easily deduce which pair has the 0. It can only be the pair corresponding to the cyclic substring that has the smallest cyclic substring of length 2^{k-1} as the second half. Logically, this pair has to be 2^{k-1} positions to the left.

   Using this technique, we can get a permutation array so that the second elements are sorted.

2. To finish, we have to sort the pairs by the first element using a stable sort algorithm. In particular, counting sort can be implemented to be stable. It is also important to note that since we are reordering the pairs twice, the permutation arrays would have to be merged. However, we can do so by providing counting sort with the permutation array from the first step.

Implementation

We first have to implement a stable counting sort. Regarding the arguments:

`T`: Types of the container to be sorted. The only requirements are that it has a size method and constant time access to any element.

`source`: Container to sort. Usually a string or a vector of integers.

`max_dif`: The function assumes that all values in source are between 0 and max_dif

`previousOrder`: If this is not the first of several sorting processes applied to the same container, this argument should be the current (before this sorting process) permutation of the container so that the result is a permutation with respect to the initial ordering of the container and not the current one.

The function returns a permutation of the container so that if the elements were placed in that order, they would be sorted.

```
20 template <typename T>
21 vector<int> counting_sort(T const source, int max_dif,
22                           vi *previousOrder = nullptr) {
23     vi cnt(max_dif);
24     vi res(source.size());
25     int n = source.size();
26
27     for (int i = 0; i < n; ++i) {
28         cnt[source[i]]++;
29     }
30     for (int i = 1; i < max_dif; ++i) {
31         cnt[i] += cnt[i - 1];
32     }
33     for (int i = n - 1; i >= 0; --i) {
34         if (previousOrder) {
35             res[--cnt[source[i]]] = previousOrder->operator[](i);
36         } else {
37             res[--cnt[source[i]]] = i;
38         }
39     }
40     return res;
41 }
```

The construction of the suffix array is follows the algorithm explained before:

```cpp
template <typename T> struct SuffixArray {
    T const s;
    int n;
    vi sa;

    int &operator[](int index) { return sa[index]; }
    int const &operator[](int index) const { return sa[index]; }

    SuffixArray(T const &s) : s(s), n(s.size()) {
        const int alphabet = 256;
        vi p, c(n);

        p = counting_sort(s, alphabet);

        int classes = 1; // At least one class (one element)
        c[p[0]] = classes - 1;
        for (int i = 1; i < n; ++i) {
            classes += s[p[i]] != s[p[i - 1]]; // We have a new class
            c[p[i]] = classes - 1;
        }

        vector<int> pn(n), cn(n);
        for (int h = 0; (1 << h) < n; ++h) {
            for (int i = 0; i < n; ++i) {
                pn[i] = (p[i] - (1 << h) + n) % n;
                // New classes after ordering by second element:
                cn[i] = c[pn[i]];
            }
            p = counting_sort(cn, classes, &pn);

            classes = 1;
            cn[p[0]] = classes - 1;
            for (int i = 1; i < n; ++i) {
                pii cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
                pii prev = {c[p[i - 1]], c[(p[i - 1] + (1 << h)) % n]};

                classes += cur != prev; // We have a new class
                cn[p[i]] = classes - 1;
            }
            swap(c, cn);
        }
        this->sa = p;
    }
```

Running time: $\mathcal{O}(n \log n)$

4.3.2 Longest common prefix (LCP)

The LCP of two suffixes can be calculated using binary search if we stored the `c` array of each iteration in the construction of the suffix array. Furthermore, it is rather simple to calculate since knowing the classes of the cyclic substrings allows us to compare blocks of size 2^i in constant time.

However, this requires $\mathcal{O}(n \log n)$ memory and there is a better option. It is possible to calculate the LCP of suffixes that are adjacent in the suffix array in linear time. Then, if we need to know the LCP of two suffixes that are not adjacent, we can use an RMQ since the answer will be the minimum common prefix encountered while going from one suffix to the other in the order of the suffix array. Furthermore, the LCP of contiguous elements is very useful in many applications.

Therefore, the goal is to build an array `lcp` where `lcp[i]` is the length of the LCP between the suffixes starting in index `sa[i]` and `sa[i+1]`. To do so, we will use Kasai's algorithm.

To simplify the notation, in the following paragraphs $s(i)$ will denote the suffix that starts on position i. Additionally, we will also need the inverse of the suffix array (sa), we will call this function $rank$. This function fulfills that $rank(sa(i)) = i$

The algorithm processes the suffixes in increasing order of starting index. That is to say, the first suffix to be processed is $s(0)$ (entire string) and, the last, $s(n-1)$. The first iteration of the algorithm is simple enough, we will just calculate $\alpha_0 = rank(0)$ the position of the first suffix in the suffix array and compare $s(0)$ with $s(\alpha_0 - 1)$.

However, in the following iterations, we cannot simply compare the entire strings since that would lead to a quadratic solution. To avoid it, we will make us of the following:

$$lcp(s(i+1)) \geq lcp(s(i)) - 1 \tag{4.1}$$

Proof

If $s(i)$ has a lcp of k_i, $s(i)$ must share its first k_i characters with $s(rank(i)-1)$ (preceding suffix in sa). Furthermore $s(i+1)$ is the result of deleting the last character of $s(i)$. If we delete the last character of $s(rank(i)-1)$ we get a suffix that has a lcp of $k_i - 1$ with $s(i+1)$. We can denote this suffix as β.

As long as k_i was greater than 0, we know that β will be before $s(i+1)$ in the suffix array since deleting a character that is equal in two suffixes cannot change their relative order. If they are adjacent, $k_i - 1$ will be the answer. Otherwise, the answer must be at least $k_i - 1$ because the elements in between β and $s(i+1)$ cannot have a lower lcp since the lcp of a range is the minimum of the lcp's of contiguous elements.

If k_i was 0, then we cannot extract any information and equation 4.1 provides a useless bound (-1).

□

What we have proved is that the lcp will decrease by at most 1 in each iteration. However, since it is bounded by n, that means that it can only increase or decrease $\mathcal{O}(n)$ times. Therefore, in each iteration we can check if the lcp has increased or not by comparing the characters without performing too many operations.

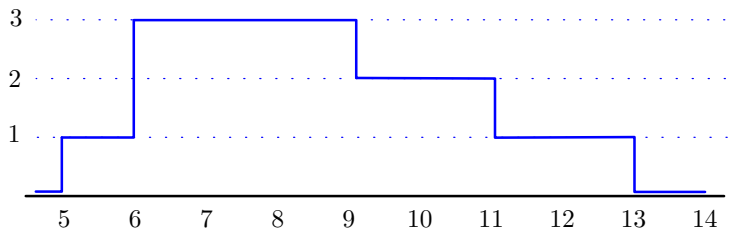
Implementation

```
87      vi computeLCP() {
88          vi rank(n, 0), lcp(n - 1, 0);
89          for (int i = 0; i < n; ++i) {
90              rank[sa[i]] = i;
91          }
92          int k = 0;
93          for (int i = 0; i < n; ++i) {
94              if (rank[i] == n - 1) {
95                  k = 0;
96                  continue;
97              }
98              int j = sa[rank[i] + 1];
99              while (i + k < n && j + k < n && s[i + k] == s[j + k]) {
100                 k++;
101             }
102             lcp[rank[i]] = k;
103             k = max(k - 1, 0);
104         }
105         return lcp;
106     }
```

Running time: $\mathcal{O}(n)$

Observation

It is important to keep in mind than when the LCP function increases the previous prefix is preserved. For instance, if the LCP function has the following shape:



- There is a prefix of lenth 1 shared between suffixes 5 and 13
- There is a prefix of lenth 2 shared between suffixes 6 and 11
- There is a prefix of lenth 3 shared between suffixes 6 and 9

4.3.3 Applications

Search for a pattern

In order to search for a patter p in a text t we can start by creating the suffix array of t in $\mathcal{O}(|t| \log |t|)$. Then, we can observe that s has to be a prefix of a suffix of t. Since we have sorted the suffixes, we can binary search them against s. This search will have a complexity of $\mathcal{O}(|s| \log |t|)$.

Furthermore, we can use a second binary search to look for the last occurrence of the string as a prefix in the suffix array.

Comparing two substrings

After precalculating the suffix array of the string, we can compare two substrings of equal length in constant time. To do so, we will have to store the class array of each iteration of the construction of the segment tree.

Those arrays enable us to compare blocks of 2^i characters in constant time. Therefore, we can split the substrings of length l into two blocks of size $2^{\lfloor \log l \rfloor}$ and compare those. We can place the blocks so that they overlap but cover the entire string.

In the previous example, for instance, we would first compare the classes that correspond to the blue blocks and, if those were equal, we would compare the classes corresponding to the green ones. Since all blocks have a length of 2–, their classes have been calculated during the construction of the suffix array.

Longest repeated substring

Having both the suffix array and the LCP array, obtaining the longest repeated substring is trivial. This string must be the prefix of (at least) two consecutive suffixes in the suffix array. Therefore, its length will be the maximum LCP between consecutive suffixes. That is to say, the maximum of the array LCP.

Longest common substring

We will start by concatenating all strings ($\{s_i\}$) separated by some characters that are smaller than the ones on the strings. It is important to make sure that these separators are different for each string and do not appear anywhere else in the string. The next step is to create both the suffix array and the LCP array of this new string (t).

The longest common substring will be a prefix of at least two suffixes of t. Furthermore, we have to make sure that we are not picking a string that appears twice in the same string. To avoid that scenario, we will make sure that the two suffixes that share the prefix start on different initial strings (s_j). If we have an array owner that tells us in which initial string does the i-th suffix in the suffix array starts, we can write the following code:

```
ans = 0
for i in [0, n-1):
    if (owner[i] ≠ owner[i+1]):
        ans = max(ans, lcp[i])
```

It is also important to note that if we keep track of the owners of each suffix, we can know in how many s_i does a particular pattern appear. This concept is used in `UVA-11107`. In that problem, we are asked to find the longest substring that appears in at least half of the initial chains.

To do so, we can binary search the length of the solution (l). Then, we can check if there is chain of that length that appears in half of the initial chains in linear time. This can be rewritten as checking that the preffix function is at least l in a range of suffixes that belong to at least $n/2$ chains. The following snippet implements this check for a given l (`target`):

```cpp
vi test(SuffixArray const &sf, vi const &lcp, vi &beginnings, int target,
        int minSize) {
    set<int> presentIn;
    vi answers;
    int lans;
    for (int i = 0; i < lcp.size(); ++i) {
        // Find the owner of this suffix
        int c1 = upper_bound(all(beginnings), sf[i]) - beginnings.begin();
        int c2 = upper_bound(all(beginnings), sf[i + 1]) - beginnings.begin();

        if (lcp[i] ≥ target) {
            // Continue the range
            if (c1 ≠ c2) {
                db(cout << c1 << " " << c2 << endl);
                presentIn.insert(c1);
                presentIn.insert(c2);
                lans = sf[i];
            }
        } else {
            // Break the range
            if (presentIn.size() > minSize) {
                answers.push_back(lans);
                echo(answers);
            }
            presentIn.clear();
        }
    }
    if (presentIn.size() > minSize) {
        answers.push_back(lans);
    }
    return answers;
}
```

In that function we used an array beginnings that stores the initial position of each chain

s_i. This array is the simplest way to implement the owner concept since it allows us to find the owner of a specific suffix in $\mathcal{O}(\log n)$ where n is the number of chains s_i.

Number of different substrings

We can count the different substrings in linear time if we have already computed both the suffix array and the LCP array.

To count the substrings, we will iterate through the suffix array and count how many prefixes of each suffix are unique. We only have to observe that all prefixes of suffix $sa(i)$ will be new except for the first $lcp(i)$ ones as these ones already occur in previous suffixes.

Therefore, the final result is:

$$\sum_{i=0}^{n-1}(n - p[i]) - \sum_{i=0}^{n-2} lcp[i] = \frac{n^2 + n}{2} - \sum_{i=0}^{n-2} lcp[i]$$

Sort cyclic shifts

Since the construction algorithm sorts all cyclic shifts of the given string, if we do not concatenate the ending character \$, we can use it to sort the cyclic shifts of the string

Chapter 5

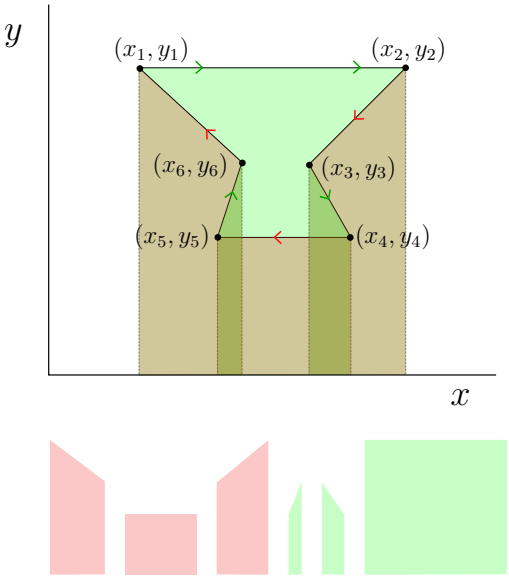
Geometry

5.1 Area of polygon

In this problem, we are given a polygon expressed as a list of vertices. We are tasked with calculating the area of the polygon.

For now, we will assume that the points are ordered in a clock-wise fashion and we will consider three different scenarios for each segment:

- $\Delta x = 0$. It does not add any area to the polygon.
- $\Delta x \neq 0$. It has an area associated with it that is equal to the area between the segment and the x axis
  - $\Delta x > 0$. We consider this area positive (green on the figure)
  - $\Delta x < 0$. We consider this area negative (red on the figure)



Now we add all the areas associated with each segment and we get a total area. If the vertices were ordered in a counter-clock-wise fashion, this total area will be negative. Therefore, we will only consider the absolute value of the result.

```
20
21  double calcAreaSegment(pii a1, pii a2) {
22      if (a1.fi == a2.fi)
23          return 0;
24
25      return (a2.fi - a1.fi) * (max(a1.se, a2.se) - abs(a1.se - a2.se) / 2.0);
26  }
```

```cpp
double calcAreaPolygon(vector<pii> points) {
    double ans = 0;
    for (int i = 1; i < points.size(); ++i) {
        ans += calcAreaSegment(points[i - 1], points[i]);
    }
    ans += calcAreaSegment(points.back(), points[0]);
    return abs(ans);
}
```

Running time: $\mathcal{O}(n)$

(n = vertices)

Chapter 6

Ad-hoc

The algorithms in this chapter do not fall into standard categories, so they ended up here.