Algo_LIb

Generated by Doxygen 1.9.8

Chapter 1

Algorithm_Library

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 DisjointSet Class Reference

`#include <DisjointSet.hpp>`

Public Member Functions

- DisjointSet (unsigned int elements)
  The constructor takes as input the amount of elements that exists.
- void unionSets (unsigned int a, unsigned int b)
  The unionSets function takes two elements as input and merges the sets that they are in.
- bool query (unsigned int a, unsigned int b)
  The query function takes two elements as input and returns true if they are in the same set. Otherwise it returns false.

4.1.1 Constructor & Destructor Documentation

4.1.1.1 DisjointSet()

```
DisjointSet::DisjointSet (
            unsigned int elements )
```

The constructor takes as input the amount of elements that exists.

4.1.2 Member Function Documentation

4.1.2.1 query()

```
bool DisjointSet::query (
            unsigned int a,
            unsigned int b )
```

The query function takes two elements as input and returns true if they are in the same set. Otherwise it returns false.

4.1.2.2 unionSets()

```
void DisjointSet::unionSets (
            unsigned int a,
            unsigned int b )
```

The unionSets function takes two elements as input and merges the sets that they are in.

The documentation for this class was generated from the following file:

- Data_Structures/Disjoint_Set/DisjointSet.hpp

4.2 Edge Struct Reference

```
#include <Bellman_Ford.hpp>
```

Public Attributes

- size_t from
- size_t to
- long long int weight
- size_t destination
- unsigned long long int travelTime
- unsigned long long int firstDeparture
- unsigned long long int repeatDepartures
- long long int capacity
- long long int flow = 0
- long long int reverse = -1

4.2.1 Member Data Documentation

4.2.1.1 capacity

```
long long int Edge::capacity
```

4.2.1.2 destination

```
size_t Edge::destination
```

4.2.1.3 firstDeparture

```
unsigned long long int Edge::firstDeparture
```

4.2.1.4 flow

```
long long int Edge::flow = 0
```

4.2.1.5 from

`size_t Edge::from`

4.2.1.6 repeatDepartures

`unsigned long long int Edge::repeatDepartures`

4.2.1.7 reverse

`long long int Edge::reverse = -1`

4.2.1.8 to

`size_t Edge::to`

4.2.1.9 travelTime

`unsigned long long int Edge::travelTime`

4.2.1.10 weight

`long long int Edge::weight`

The documentation for this struct was generated from the following files:

- Graphs/Bellman_Ford/Bellman_Ford.hpp
- Graphs/Dijkstra_Time_Table/shortest_path_time_table.hpp
- Graphs/Maximum_Flow/Maximum_Flow.hpp

4.3 FenwickTree Class Reference

`#include <FenwickTree.hpp>`

Public Member Functions

- FenwickTree (size_t initSize)
- ∼FenwickTree ()
- void Update (size_t index, long long int value)
- long long int Query (size_t index)

4.3.1 Constructor & Destructor Documentation

4.3.1.1 FenwickTree()

```
FenwickTree::FenwickTree (
            size_t initSize )
```

4.3.1.2 ∼FenwickTree()

```
FenwickTree::~FenwickTree ( )
```

4.3.2 Member Function Documentation

4.3.2.1 Query()

```
long long int FenwickTree::Query (
            size_t index )
```

4.3.2.2 Update()

```
void FenwickTree::Update (
            size_t index,
            long long int value )
```

The documentation for this class was generated from the following file:

- Data_Structures/FenwickTree/FenwickTree.hpp

4.4 Point< T > Struct Template Reference

```
#include <Points.hpp>
```

Public Member Functions

- double length ()
- double magnitude ()
- T lengthSquared ()
- T magnitudeSquared ()
- bool operator== (Point< T > const &other)
- bool operator!= (Point< T > const &other)
- Point< T > operator+ (Point< T > const &other)
- Point< T > operator- (Point< T > const &other)

Public Attributes

- T x
- T y

4.4.1 Member Function Documentation

4.4.1.1 length()

```
template<typename T >
double Point< T >::length ( )  [inline]
```

4.4.1.2 lengthSquared()

```
template<typename T >
T Point< T >::lengthSquared ( )  [inline]
```

4.4.1.3 magnitude()

```
template<typename T >
double Point< T >::magnitude ( )  [inline]
```

4.4.1.4 magnitudeSquared()

```
template<typename T >
T Point< T >::magnitudeSquared ( )  [inline]
```

4.4.1.5 operator"!=()

```
template<typename T >
bool Point< T >::operator!= (
            Point< T > const & other )  [inline]
```

4.4.1.6 operator+()

```
template<typename T >
Point< T > Point< T >::operator+ (
            Point< T > const & other )  [inline]
```

4.4.1.7 operator-()

```
template<typename T >
Point< T > Point< T >::operator- (
            Point< T > const & other )  [inline]
```

4.4.1.8 operator==()

```
template<typename T >
bool Point< T >::operator== (
            Point< T > const & other ) [inline]
```

4.4.2 Member Data Documentation

4.4.2.1 x

```
template<typename T >
T Point< T >::x
```

4.4.2.2 y

```
template<typename T >
T Point< T >::y
```

The documentation for this struct was generated from the following file:

- Geometry/Points/Points.hpp

4.5 Prime_Sieve Class Reference

```
#include <Sieve.hpp>
```

Public Member Functions

- const std::vector< size_t > & get_Primes ()
- Prime_Sieve (size_t n)
- size_t get_Number_Of_Primes ()
- bool is_Prime (size_t i)

4.5.1 Constructor & Destructor Documentation

4.5.1.1 Prime_Sieve()

```
Prime_Sieve::Prime_Sieve (
            size_t n )
```

4.5.2 Member Function Documentation

4.5.2.1 get_Number_Of_Primes()

```
size_t Prime_Sieve::get_Number_Of_Primes ( )
```

4.5.2.2 get_Primes()

```
const std::vector< size_t > & Prime_Sieve::get_Primes ( )
```

4.5.2.3 is_Prime()

```
bool Prime_Sieve::is_Prime (
            size_t i )
```

The documentation for this class was generated from the following file:

- Prime_Numbers/Prime_Sieve/Sieve.hpp

4.6 Rational< T > Struct Template Reference

```
#include <rational.hpp>
```

Public Member Functions

- Rational< T > operator+ (const Rational< T > &other) const
- Rational< T > operator- (const Rational< T > &other) const
- Rational< T > operator* (const Rational< T > &other) const
- Rational< T > operator/ (const Rational< T > &other) const
- void normalise ()

Public Attributes

- T numerator
- T denominator

4.6.1 Member Function Documentation

4.6.1.1 normalise()

```
template<typename T >
void Rational< T >::normalise ( )
```

4.6.1.2 operator*()

```
template<typename T >
Rational< T > Rational< T >::operator* (
            const Rational< T > & other ) const
```

4.6.1.3 operator+()

```
template<typename T >
Rational< T > Rational< T >::operator+ (
             const Rational< T > & other ) const
```

4.6.1.4 operator-()

```
template<typename T >
Rational< T > Rational< T >::operator- (
             const Rational< T > & other ) const
```

4.6.1.5 operator/()

```
template<typename T >
Rational< T > Rational< T >::operator/ (
             const Rational< T > & other ) const
```

4.6.2 Member Data Documentation

4.6.2.1 denominator

```
template<typename T >
T Rational< T >::denominator
```

4.6.2.2 numerator

```
template<typename T >
T Rational< T >::numerator
```

The documentation for this struct was generated from the following file:

- Arithmetic/Rational arithmetic/rational.hpp

4.7 Segment_Tree< T > Class Template Reference

```
#include <Segment_Tree.hpp>
```

Public Member Functions

- Segment_Tree (const std::vector< T > &v, std::function< T(T, T)> mergeFunc)
- T Query (int left, int right)

4.7.1 Constructor & Destructor Documentation

4.7.1.1 Segment_Tree()

```
template<class T >
Segment_Tree< T >::Segment_Tree (
            const std::vector< T > & v,
            std::function< T(T, T)> mergeFunc )  [inline]
```

4.7.2 Member Function Documentation

4.7.2.1 Query()

```
template<class T >
T Segment_Tree< T >::Query (
            int left,
            int right )  [inline]
```

The documentation for this class was generated from the following file:

- Data_Structures/Segment_Tree/Segment_Tree.hpp

4.8 SuffixArray Class Reference

```
#include <Suffix_Sorting.hpp>
```

Public Member Functions

- SuffixArray (std::string const &s)
- size_t getSuffix (size_t i)

4.8.1 Constructor & Destructor Documentation

4.8.1.1 SuffixArray()

```
SuffixArray::SuffixArray (
            std::string const & s )
```

4.8.2 Member Function Documentation

4.8.2.1 getSuffix()

```
size_t SuffixArray::getSuffix (
            size_t i )
```

The documentation for this class was generated from the following file:

- Strings/Suffix-sorting/Suffix_Sorting.hpp

4.9 TrieAutomaton Class Reference

```
#include <Aho-Corasick.hpp>
```

Public Member Functions

- TrieAutomaton ()
- ∼TrieAutomaton ()
- void add_string (std::string const &s)
- void construct_automaton ()
- std::vector< std::vector< size_t > > search (std::string const &s)

4.9.1 Constructor & Destructor Documentation

4.9.1.1 TrieAutomaton()

```
TrieAutomaton::TrieAutomaton ( )
```

4.9.1.2 ∼TrieAutomaton()

```
TrieAutomaton::~TrieAutomaton ( )
```

4.9.2 Member Function Documentation

4.9.2.1 add_string()

```
void TrieAutomaton::add_string (
            std::string const & s )
```

4.9.2.2 construct_automaton()

```
void TrieAutomaton::construct_automaton ( )
```

4.9.2.3 search()

```
std::vector< std::vector< size_t > > TrieAutomaton::search (
            std::string const & s )
```

The documentation for this class was generated from the following file:

- Strings/Aho-corasick/Aho-Corasick.hpp

Chapter 5

File Documentation

5.1 Arithmetic/Chinese-Remainder-Theorem/Chinese_Remainder.hpp File Reference

```
#include <vector>
#include <utility>
#include "Arithmetic\Modular-Arithmetic\modular.hpp"
```

Functions

- long long int Solve_System_Of_Congruencies (std::vector< std::pair< long long int, long long int > > equations, bool co_prime=false)

5.1.1 Function Documentation

5.1.1.1 Solve_System_Of_Congruencies()

```
long long int Solve_System_Of_Congruencies (
            std::vector< std::pair< long long int, long long int > > equations,
            bool co_prime = false )
```

Function to solve a system of congruencies of the form

x = a1 mod b1 x = a2 mod b2x = an mod bn

Takes as argument a vector of pairs, where the i:th pair corresponds to (ai, bi)

b1-bn do not have to be co-prime //ACTUALLY THEY DO, BUT THIS WILL BE FIXED IN THE FUTURE If you know b1-bn are co_prime then setting the flag co_prime to true may improve performance

Returns -1 if there is no solution

5.2 Chinese_Remainder.hpp

Go to the documentation of this file.

```
#ifndef CHINESE_REMAINDER_HPP
#define CHINESE_REMAINDER_HPP
#include<vector>
#include<utility>
#include"Arithmetic\Modular-Arithmetic\modular.hpp"

long long int Solve_System_Of_Congruencies(std::vector<std::pair<long long int, long long int»
equations, bool co_prime = false){

    if(co_prime) goto start_solving;



    start_solving:

    long long int lcm = 1;
    for(auto& pair : equations){
        lcm *= pair.second;
    }

    long long int ans = 0;

    for(auto& pair : equations){
        long long int Mi = lcm/pair.second;
        long long int xi = Mi % pair.second;
        long long int bi = modInverse(xi, pair.second);
        long long int yi = (bi * Mi) % lcm;
        ans = modAdd(ans, modMult(pair.first, yi, lcm), lcm);
    }

    return ans;
}

#endif
```

5.3 Arithmetic/Modular-Arithmetic/modular.hpp File Reference

```
#include <utility>
#include <cstdlib>
#include <stddef.h>
```

Typedefs

- typedef long long int ll
- typedef unsigned long long int ull
- typedef __uint128_t u128
- typedef __int128_t i128

Functions

- ll mod (const i128 a, const i128 m)
- ll modMult (const i128 a, const i128 b, const i128 m)
- ll modAdd (const i128 a, const i128 b, const i128 m)
- ll modSub (const i128 a, const i128 b, const i128 m)
- i128 gcd (i128 a, i128 b, i128 &x0, i128 &y0)
- i128 modInverse (const i128 b, const i128 m)
- ll modDiv (const i128 a, const i128 b, const i128 m)

5.3.1 Typedef Documentation

5.3.1.1 i128

```
typedef __int128_t i128
```

5.3.1.2 ll

```
typedef long long int ll
```

5.3.1.3 u128

```
typedef __uint128_t u128
```

5.3.1.4 ull

```
typedef unsigned long long int ull
```

5.3.2 Function Documentation

5.3.2.1 gcd()

```
i128 gcd (
            i128 a,
            i128 b,
            i128 & x0,
            i128 & y0 )
```

5.3.2.2 mod()

```
ll mod (
            const i128 a,
            const i128 m )  [inline]
```

5.3.2.3 modAdd()

```
ll modAdd (
            const i128 a,
            const i128 b,
            const i128 m )  [inline]
```

5.3.2.4 modDiv()

```
ll modDiv (
            const i128 a,
            const i128 b,
            const i128 m )
```

5.3.2.5 modInverse()

```
i128 modInverse (
            const i128 b,
            const i128 m )
```

5.3.2.6 modMult()

```
ll modMult (
            const i128 a,
            const i128 b,
            const i128 m ) [inline]
```

5.3.2.7 modSub()

```
ll modSub (
            const i128 a,
            const i128 b,
            const i128 m ) [inline]
```

5.4 modular.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#include<utility>
#include<cstdlib>
#include<stddef.h>


typedef long long int ll;
typedef unsigned long long int ull;
typedef __uint128_t u128;
typedef __int128_t i128;

//Returnerar a mod m. Fungerar för både negativa och positiva tal a.
inline ll mod(const i128 a, const i128 m){
    return (m+(a%m))%m;
}

//Returnerar (a*b) mod m. Där a,b,m får plats i ett 64-bitars tal
inline ll modMult(const i128 a, const i128 b, const i128 m){
    return mod(a * b, m);
}

//Returnerar (a+b) mod m. Där a,b,m får plats i ett 64-bitars tal
inline ll modAdd(const i128 a, const i128 b, const i128 m){
    return mod(a + b, m);
}

//Returnerar (a-b) mod m. Där a,b,m får plats i ett 64-bitars tal
```

```
inline ll modSub(const i128 a, const i128 b, const i128 m){
    return mod(a - b, m);
}

//Returnerar största gemensamma nämnaren till a och b
i128 gcd(i128 a, i128 b, i128& x0, i128& y0){
    a = std::abs(a);
    b = std::abs(b);
    if(a > b){
        std::swap(a,b);
    }

    if (a == 0){
        x0 = 0;
        y0 = 1;
        return b;
    }
    i128 x1, y1, d;
    d = gcd(b % a, a, x1, y1);

    x0 = y1 - (b/a) * x1;
    y0 = x1;

    return d;
}

//Returnerar inversen till b i Z_m
i128 modInverse(const i128 b, const i128 m){
    i128 x, y, d;
    d = gcd(b, m, x, y);

    if(d != 1) return -1;

    return mod(x, m);
}

//Returnerar a * b^(-1) i Z_m
ll modDiv(const i128 a, const i128 b, const i128 m){
    i128 inverse = modInverse(b, m);
    if(inverse != -1){
        return modMult(a, inverse, m);
    }
    else{
        return -1;
    }
}
```

5.5 Arithmetic/Rational arithmetic/rational.hpp File Reference

```
#include <iostream>
#include <utility>
```

Classes

- struct Rational< T >

Functions

- template<typename T >
  T gcd (T a, T b)
- template<typename T >
  std::istream & operator>> (std::istream &is, Rational< T > &m)
- template<typename T >
  std::ostream & operator<< (std::ostream &os, const Rational< T > &m)

5.5.1 Function Documentation

5.5.1.1 gcd()

```
template<typename T >
T gcd (
            T a,
            T b )
```

5.5.1.2 operator<<()

```
template<typename T >
std::ostream & operator<< (
            std::ostream & os,
            const Rational< T > & m )
```

5.5.1.3 operator>>()

```
template<typename T >
std::istream & operator>> (
            std::istream & is,
            Rational< T > & m )
```

5.6 rational.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#include<iostream>
#include<utility>

//A datatype that stores and performs operations on fractions. T must implement the +,-,*,/,«,» operators.
//In addition to that, euclids algorithm needs to work for T. It must also be possible to use T with the std::abs function.
template<typename T>
struct Rational{
    T numerator, denominator;

    Rational<T> operator+(const Rational<T>& other) const;
    Rational<T> operator-(const Rational<T>& other) const;
    Rational<T> operator*(const Rational<T>& other) const;
    Rational<T> operator/(const Rational<T>& other) const;

    void normalise();
};

template<typename T>
T gcd(T a, T b);

template<typename T>
std::istream & operator»(std::istream & is, Rational<T>& m);

template<typename T>
std::ostream & operator«(std::ostream & os, const Rational<T> & m);

template<typename T>
Rational<T> Rational<T>::operator+(const Rational<T>& other) const{
    Rational<T> sum;
    sum.denominator = denominator * other.denominator;
    sum.numerator = numerator * other.denominator + other.numerator * denominator;

```

```
    sum.normalise();
    return sum;
}

template<typename T>
Rational<T> Rational<T>::operator-(const Rational<T>& other) const{
    Rational<T> difference;
    difference.denominator = denominator * other.denominator;
    difference.numerator = numerator * other.denominator - other.numerator * denominator;

    difference.normalise();
    return difference;
}

template<typename T>
Rational<T> Rational<T>::operator*(const Rational<T>& other) const{
    Rational<T> product;
    product.denominator = denominator * other.denominator;
    product.numerator = numerator * other.numerator;

    product.normalise();
    return product;
}

template<typename T>
Rational<T> Rational<T>::operator/(const Rational<T>& other) const{
    Rational<T> swapped;
    swapped.denominator = other.numerator;
    swapped.numerator = other.denominator;

    Rational<T> ans = *(this) * swapped;
    ans.normalise();
    return ans;
}

template<typename T>
void Rational<T>::normalise(){
    T divisor = gcd(denominator, numerator);
    denominator /= divisor;
    numerator /= divisor;

    if(denominator < 0 && numerator < 0){
        denominator = -denominator;
        numerator = -numerator;
    }
    else if(denominator < 0){
        denominator = -denominator;
        numerator = -numerator;
    }
}

template<typename T>
std::istream & operator»(std::istream & is, Rational<T>& m){
    is » m.numerator » m.denominator;
    m.normalise();
    return is;
}

template<typename T>
std::ostream & operator«(std::ostream & os, const Rational<T> & m){
    os « m.numerator « " / " « m.denominator;
    return os;
}

template<typename T>
T gcd(T a, T b){
    a = std::abs(a);
    b = std::abs(b);
    if(a > b){
        std::swap(a,b);
    }

    if (a == 0)
        return b;
    return gcd(b % a, a);
}

```

5.7 Data_Structures/Disjoint_Set/DisjointSet.hpp File Reference

```
#include <vector>
#include <functional>
```

Classes

- class DisjointSet

5.8 DisjointSet.hpp

Go to the documentation of this file.

```
/*
    AUTHOR: Oliver Lindgren
*/

#ifndef DISJOINTSET_HPP
#define DISJOINTSET_HPP

#include<vector>
#include<functional>

/*
A class designed to keep track of N different sets of N total elements.
Has a function to merge two sets, and a function to check if two elements are in the same set.
TODO: Change to size_t
*/
class DisjointSet {
private:
    std::vector<int> comp;

    int representative(unsigned int a);

public:
    DisjointSet(unsigned int elements);

    void unionSets(unsigned int a, unsigned int b);

    bool query(unsigned int a, unsigned int b);
};


DisjointSet::DisjointSet(unsigned int elements) : comp(elements, -1) {};

void DisjointSet::unionSets(unsigned int a, unsigned int b){
    a = representative(a);
    b = representative(b);

    if(a == b) return;

    if(comp[a] > comp[b]) std::swap(a,b);

    comp[a] += comp[b]; //The size of set containing a has been increased by size of set containing b.
    comp[b] = a; //All elements that had b as a representative will now have a as a representative.
}

int DisjointSet::representative(unsigned int a){
    if(comp[a] < 0) return a;

    //recursively find and update our representative
    comp[a] = representative(comp[a]);
    return comp[a];
}

bool DisjointSet::query(unsigned int a, unsigned int b){
    return representative(a) == representative(b);
```

```
}



#endif
```

5.9 Data_Structures/FenwickTree/FenwickTree.hpp File Reference

```
#include <cstddef>
```

Classes

- class FenwickTree

5.10 FenwickTree.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef FENWICKTREE_HPP
#define FENWICKTREE_HPP

#include<cstddef>

//A class designed to compute and maintain the prefix sum of an array.
class FenwickTree{
private:
    long long int* base;
    size_t size;
public:
    //initSize is the size of the array to be stored.
    FenwickTree(size_t initSize);
    ~FenwickTree();

    //This increases the value of array[index] with value.
    void Update(size_t index, long long int value);

    //This querys the array to get the sum of all the values in the array in the range [0, index)
    long long int Query(size_t index);
};
```

```
FenwickTree::FenwickTree(size_t initSize){
    size = initSize+1;
    base = new long long int[size]();
}

FenwickTree::~FenwickTree(){
    delete[] base;
}


void FenwickTree::Update(size_t index, long long int value){
    index++;

    //Use bit indexing to update necessary values.
    while(index < size){
        base[index] += value;
        index += index & (-index);
    }
}

long long int FenwickTree::Query(size_t index){
    long long int sum = 0;

    //Use bit indexing to retrieve necessary values.
    while(index > 0){
        sum += base[index];
        index -= index & (-index);
    }
    return sum;
}
#endif
```

5.11 Data_Structures/Segment_Tree/Segment_Tree.hpp File Reference

```
#include <vector>
#include <functional>
```

Classes

- class Segment_Tree< T >

5.12 Segment_Tree.hpp

Go to the documentation of this file.

```
#ifndef SEGMENT_TREE_HPP
#define SEGMENT_TREE_HPP

#include<vector>
#include <functional>

template<class T>
class Segment_Tree{

int leftMin, rightMax;
std::vector<T> internal;
std::function<T(T,T)> merge;

void compute(int at, int l, int r, const std::vector<T>& v){
    if(l == r){
```

```
        internal[at] = v[l];
        return;
    }
    int mid = (l+r)/2;
    compute(at*2 + 1, l, mid, v);
    compute(at*2 + 2, mid+1, r, v);

    internal[at] = merge(internal[at*2+1], internal[at*2+2]);
}

T search(int at, int l, int r, int currL, int currR){
    if(currL >= l && currR <= r) return internal[at];

    int mid = (currL + currR) / 2;
    if(r <= mid) return search(at*2+1, l, r, currL, mid);
    else if(l >= mid+1) return search(at*2+2, l, r, mid+1, currR);

    return merge(search(at*2+1, l, r, currL, mid), search(at*2+2, l, r, mid+1, currR));
}

public:

Segment_Tree(const std::vector<T>& v, std::function<T(T,T)> mergeFunc){
    internal.resize(v.size()*4);
    merge = mergeFunc;
    leftMin = 0;
    rightMax = v.size()-1;
    compute(0, leftMin, rightMax, v);
}

T Query(int left, int right){
    return search(0, left, right, leftMin, rightMax);
}

};

#endif
```

5.13 Geometry/Convex_Hull/Convex_Hull.hpp File Reference

```
#include "Geometry\Points\Points.hpp"
#include <vector>
#include <algorithm>
```

Functions

- template<typename T >
  std::vector< Point< T > > convex_hull (std::vector< Point< T > > points)

5.13.1 Function Documentation

5.13.1.1 convex_hull()

```
template<typename T >
std::vector< Point< T > > convex_hull (
            std::vector< Point< T > > points )
```

5.14 Convex_Hull.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef CONVEX_HULL_HPP
#define CONVEX_HULL_HPP
#include"Geometry\Points\Points.hpp"
#include<vector>
#include<algorithm>

/*
Function that takes a vector of points and returns the convex hull encircling them.
The hull is given as a list of points in counterclockwise order, each point being a node in the hull.
If the points are all colinear then the two endpoints are given.

Time Complexity: O(N log(n))
*/

template<typename T>
std::vector<Point<T» convex_hull(std::vector<Point<T>> points){
    int leftMostPoint = 0;
    Point<T> org = {0,0};
    std::vector<Point<T» hull;

    //This is to later remove duplicates
    std::sort(points.begin(), points.end(), [&org](Point<T> const & a, Point<T> const & b){
        if(a.x == b.x)
            return a.y < b.y;
        else
            return a.x < b.x;
    });

    //remove duplicates
    std::vector<Point<T» newPoints;
    newPoints.push_back(points[0]);
    for(int i = 1; i < points.size(); i++){
        if(points[i] != *(newPoints.rbegin())) newPoints.push_back(points[i]);
    }


    points.swap(newPoints);

    //Find the leftmost point that is definitely on the hull
    for(int i = 1; i < points.size(); i++){
        if(points[i].x < points[leftMostPoint].x) leftMostPoint = i;
        else if(points[i].x == points[leftMostPoint].x && points[i].y > points[leftMostPoint].y) leftMostPoint = i;
    }

    org = points[leftMostPoint];
    hull.push_back(points[leftMostPoint]);
    points.erase(points.begin() + leftMostPoint);

    //Sort the points based on angle from the previously determined point
    std::sort(points.begin(), points.end(), [&org](Point<T> const & a, Point<T> const & b){
        T cross = cross_product(subtract(a, org), subtract(b, org));
        if(cross == 0) return distSquared(org, a) < distSquared(org, b); //De är ko-linjära
        else return cross > 0;
    });

    //add points to the hull
    for(Point<T> & p : points){

        //if we are doing a rightturn pop last point of hull.
        while(hull.size() >= 2 && cross_product(subtract(hull[hull.size()-1], hull[hull.size()-2]), subtract(p, hull[hull.size()-2])) <= 0) hull.pop_back();
        hull.push_back(p);
    }
    return hull;
}

#endif
```

5.15 Geometry/Inside_Polygon/Inside.hpp File Reference

```
#include "Geometry\Points\Points.hpp"
#include <vector>
```

```
#include <stdexcept>
#include <utility>
#include <stddef.h>
#include <iostream>
#include <cmath>
```

Enumerations

- enum Status { Outside , OnEdge , Inside }

Functions

- double myAbs (double y)
- template<typename T >
  bool onSegment (std::pair< Point< T >, Point< T > > segment, Point< T > q)
- template<typename T >
  Status inside_poly (std::vector< Point< T > > const &polygon, Point< T > const &p)

5.15.1 Enumeration Type Documentation

5.15.1.1 Status

```
enum Status
```

Enumerator

Outside	
OnEdge	
Inside	

5.15.2 Function Documentation

5.15.2.1 inside_poly()

```
template<typename T >
Status inside_poly (
            std::vector< Point< T > > const & polygon,
            Point< T > const & p )
```

5.15.2.2 myAbs()

```
double myAbs (
            double y )
```

5.15.2.3 onSegment()

```
template<typename T >
bool onSegment (
            std::pair< Point< T >, Point< T > > segment,
            Point< T > q )
```

5.16 Inside.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef POLYGON_POINT_INSIDE_HPP
#define POLYGON_POINT_INSIDE_HPP
#include"Geometry\Points\Points.hpp"
#include<vector>
#include<stdexcept>
#include<utility>
#include<stddef.h>
#include<iostream>
#include<cmath>

enum Status{Outside, OnEdge, Inside};

//For some reason the normal abs function didn't work
double myAbs(double y){
    if(y < 0) return -y;
    else return y;
}


/*
Function to determine if a point lies on a segment.
Takes as input a pair containing the two endPoints of the segment, as well as a point.
Returns true if the point lies on the segment.

Should probably take some sort of epsilon as a value too
*/
template <typename T>
bool onSegment(std::pair<Point<T>, Point<T>> segment, Point<T> q){
    auto x = dist(segment.first, segment.second);
    auto a = dist(q, segment.first);
    auto b = dist(q, segment.second);
    auto y = x - a - b;
    double t = myAbs(y); //For some reason abs(y) returned 0 even when y == 0.7.... something
    return t < 0.00001;
}

/*
Function to determine if a point is inside, outside, or on the edge of a simple polygon.
Takes as input a vector containing the points of the simple polygon and a point p.
Returns a Status enum.
If the point is inside the polygon Status::Inside is returned.
If the point is outside the polygon Status::Outside is returned.
If the point is on the edge of the polygon Status::OnEdge is returned.
Time complexity: O(N)
*/
template <typename T>
Status inside_poly(std::vector<Point<T>> const & polygon, Point<T> const & p){
    double angleSum = 0;
    size_t n = polygon.size();
    for(size_t i = 0; i < n; i++){
        if(onSegment({polygon[i], polygon[(i+1)%n]}, p)){
            std::cerr « i « ' ' « ((i+1)%n) « '\n';
            return OnEdge;
        }
        angleSum += angle(subtract(polygon[i], p), subtract(polygon[(i+1)%n], p));
    }


    if(abs(angleSum) < 0.00001) return Outside;
    if(abs(abs(angleSum) - 2.0*(3.141592)) < 00001) return Inside;

    throw std::logic_error("Can't determine point position");
}

#endif
```

5.17 Geometry/Line_Intersection/intersection.hpp File Reference

```
#include "Geometry\Points\Points.hpp"
#include <vector>
#include <utility>
```

Functions

- template<typename T >
  std::vector< std::pair< Point< double >, Point< double > > > line_intersection (Point< T > p1, Point< T > p2, Point< T > q1, Point< T > q2)

5.17.1 Function Documentation

5.17.1.1 line_intersection()

```
template<typename T >
std::vector< std::pair< Point< double >, Point< double > > > line_intersection (
            Point< T > p1,
            Point< T > p2,
            Point< T > q1,
            Point< T > q2 )
```

5.18 intersection.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef LINE_INTERSECTION_HPP
#define LINE_INTERSECTION_HPP

#include"Geometry\Points\Points.hpp"
#include <vector>
#include<utility>

template<typename T>
std::vector<std::pair<Point<double>, Point<double>» line_intersection(Point<T> p1,Point<T> p2,
Point<T> q1, Point<T> q2){
    Point<T> deltaP = subtract(p2, p1);
    Point<T> deltaQ = subtract(q2, q1);

    T cross = cross_product(deltaP, deltaQ);
    if(cross == 0){
        Point<T> x = subtract(q1, p1);
        if(cross_product(x, deltaP) == 0) return{p1, p2};
        else return {};
    }
    else{
        double s = (double) cross_product(subtract(q1, p1), deltaP) / (double)cross;
        double t = (double) cross_product(subtract(p1, q1), deltaQ) / (double)cross;
    }
}


#endif
```

5.19 Geometry/Points/Points.hpp File Reference

```
#include <cmath>
#include <iostream>
#include <vector>
```

Classes

- struct Point< T >

Functions

- template<typename T >
  std::ostream & operator<< (std::ostream &out, Point< T > &point)
- template<typename T >
  std::istream & operator>> (std::istream &in, Point< T > &point)
- template<typename T >
  Point< T > add (Point< T > const &a, Point< T > const &b)
- template<typename T >
  Point< T > subtract (Point< T > const &a, Point< T > const &b)
- template<typename T >
  Point< T > scalar_multiplication (Point< T > const &point, T scalar)
- template<typename T >
  T dot_product (Point< T > const &a, Point< T > const &b)
- template<typename T >
  T cross_product (Point< T > const &a, Point< T > const &b)
- template<typename T >
  double length (Point< T > const &a)
- template<typename T >
  double lengthSquared (Point< T > const &a)
- template<typename T >
  double dist (Point< T > const &a, Point< T > const &b)
- template<typename T >
  double distSquared (Point< T > const &a, Point< T > const &b)
- template<typename T >
  double angle (Point< T > const &a, Point< T > const &b)
- template<typename T >
  bool intersect (Point< T > const &p1, Point< T > const &p2, Point< T > const &q1, Point< T > const &q2)
- template<typename T >
  bool simple_polygon (std::vector< Point< T > > const &v)

5.19.1 Function Documentation

5.19.1.1 add()

```
template<typename T >
Point< T > add (
          Point< T > const & a,
          Point< T > const & b )
```

5.19.1.2 angle()

```
template<typename T >
double angle (
            Point< T > const & a,
            Point< T > const & b )
```

5.19.1.3 cross_product()

```
template<typename T >
T cross_product (
            Point< T > const & a,
            Point< T > const & b )
```

5.19.1.4 dist()

```
template<typename T >
double dist (
            Point< T > const & a,
            Point< T > const & b )
```

5.19.1.5 distSquared()

```
template<typename T >
double distSquared (
            Point< T > const & a,
            Point< T > const & b )
```

5.19.1.6 dot_product()

```
template<typename T >
T dot_product (
            Point< T > const & a,
            Point< T > const & b )
```

5.19.1.7 intersect()

```
template<typename T >
bool intersect (
            Point< T > const & p1,
            Point< T > const & p2,
            Point< T > const & q1,
            Point< T > const & q2 )
```

5.19.1.8 length()

```
template<typename T >
double length (
            Point< T > const & a )
```

5.19.1.9 lengthSquared()

```
template<typename T >
double lengthSquared (
            Point< T > const & a )
```

5.19.1.10 operator<<()

```
template<typename T >
std::ostream & operator<< (
            std::ostream & out,
            Point< T > & point )
```

5.19.1.11 operator>>()

```
template<typename T >
std::istream & operator>> (
            std::istream & in,
            Point< T > & point )
```

5.19.1.12 scalar_multiplication()

```
template<typename T >
Point< T > scalar_multiplication (
            Point< T > const & point,
            T scalar )
```

5.19.1.13 simple_polygon()

```
template<typename T >
bool simple_polygon (
            std::vector< Point< T > > const & v )
```

5.19.1.14 subtract()

```
template<typename T >
Point< T > subtract (
            Point< T > const & a,
            Point< T > const & b )
```

5.20 Points.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef POINTS_HPP
#define POINTS_HPP
#include<cmath>
#include<iostream>
#include<vector>

template<typename T>
struct Point{
    T x, y;

    double length(){
        return sqrt(x*x + y*y);
    }

    double magnitude(){
        return length();
    }

    T lengthSquared(){
        return x*x + y*y;
    }

    T magnitudeSquared(){
        return lengthSquared();
    }

    bool operator==(Point<T> const & other){
        return x == other.x && y == other.y;
    }

    bool operator!=(Point<T> const & other){
        return x != other.x || y != other.y;
    }

    Point<T> operator+(Point<T> const & other){
        return {x + other.x, y + other.y};
    }

    Point<T> operator-(Point<T> const & other){
        return {x - other.x, y - other.y};
    }
};

template<typename T>
std::ostream& operator« (std::ostream& out, Point<T>& point){
    out « point.x « ' ' « point.y;
    return out;
}

template<typename T>
std::istream& operator» (std::istream& in, Point<T>& point){
    in » point.x » point.y;
    return in;
}

template<typename T>
Point<T> add(Point<T> const & a, Point<T> const & b){
    return {a.x + b.x, a.y + b.y};
}

template<typename T>
Point<T> subtract(Point<T> const & a, Point<T> const & b){
    return {a.x - b.x, a.y - b.y};
}

template<typename T>
Point<T> scalar_multiplication(Point<T> const & point, T scalar){
    return {point.x * scalar, point.y * scalar};
}

template<typename T>
T dot_product(Point<T> const & a, Point<T> const & b){
    return a.x * b.x + a.y * b.y;
}

template<typename T>
T cross_product(Point<T> const & a, Point<T> const & b){
    return a.x * b.y - b.x * a.y;
```

```
}

template<typename T>
double length(Point<T> const & a){
    return sqrt(a.x*a.x + a.y*a.y);
}

template<typename T>
double lengthSquared(Point<T> const & a){
    return a.x*a.x + a.y*a.y;
}

template<typename T>
double dist(Point<T> const & a, Point<T> const & b){
    return length(subtract(a, b));
}

template<typename T>
double distSquared(Point<T> const & a, Point<T> const & b){
    return lengthSquared(subtract(a, b));
}

template<typename T>
double angle(Point<T> const & a, Point<T> const & b){
    return atan2(cross_product(a,b), dot_product(a,b));
}

template<typename T>
bool intersect(Point<T> const & p1, Point<T> const & p2, Point<T> const & q1, Point<T> const & q2){
    return (((((p1.x-q1.x) * (p2.y - q1.y)) - ((p2.x-q1.x) * (p1.y - q1.y)) > 0) == //First determinant
                (((p1.x-q2.x) * (p2.y - q2.y)) - ((p2.x-q2.x) * (p1.y - q2.y)) < 0)) && //Second
               ((((q1.x-p1.x) * (q2.y - p1.y)) - ((q2.x-p1.x) * (q1.y - p1.y)) > 0) == //Third determinant
                (((q1.x-p2.x) * (q2.y - p2.y)) - ((q2.x-p2.x) * (q1.y - p2.y)) < 0))
    );
}
/* Source that this works
https://stackoverflow.com/a/3842157
http://www.cs.cmu.edu/~quake/robust.html
*/

template<typename T>
bool simple_polygon(std::vector<Point<T>> const & v){
    size_t n = v.size();
    bool ans = true;
    for(int i = 2; i < n; i++){
        for(int j = (i == 6 ? 1 : 0); j < i-1; j++){
            ans &= !intersect(v[i], v[(i+1)%n], v[j], v[(j+1)%n]);
        }
    }
    return ans;
}

#endif
```

5.21 Geometry/Polygon_Area/Area.hpp File Reference

```
#include "Geometry\Points\Points.hpp"
#include <vector>
#include <stddef.h>
```

Functions

- template< typename T >
  double Polygon_area (std::vector< Point< T > > points)

5.21.1 Function Documentation

5.21.1.1 Polygon_area()

```
template<typename T >
double Polygon_area (
            std::vector< Point< T > > points )
```

5.22 Area.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef POLYGON_AREA_HPP
#define POLYGON_AREA_HPP
#include"Geometry\Points\Points.hpp"
#include<vector>
#include<stddef.h>

/*
Computes the area of a simple polygon.
Takes as input a vector of points, the points of the polygon.
Returns the area of that polygon
Area is positive if points are given in counter-clockwise order.
Area is negative if points are given in clockwise order.

Time complexity: O(N)
*/
template<typename T>
double Polygon_area(std::vector<Point<T>> points){
    T area = 0;
    size_t n = points.size();
    for(size_t i = 0; i < n; i++){
        area += cross_product(points[i], points[(i+1)%n]);
    }
    return double(area) / 2.0;
}

#endif
```

5.23 Graphs/Bellman_Ford/Bellman_Ford.hpp File Reference

```
#include <vector>
#include <utility>
#include <limits>
#include <stddef.h>
```

Classes

- struct Edge

Functions

- std::vector< std::pair< long long int, size_t > > Bellman_Ford (const size_t nodes, const size_t startNode, const std::vector< Edge > &edges)

Variables

- const long long int inf = std::numeric_limits<long long int>::max()
- const long long int negInf = std::numeric_limits<long long int>::min()

5.23.1 Function Documentation

5.23.1.1 Bellman_Ford()

```
std::vector< std::pair< long long int, size_t > > Bellman_Ford (
            const size_t nodes,
            const size_t startNode,
            const std::vector< Edge > & edges )
```

5.23.2 Variable Documentation

5.23.2.1 inf

```
const long long int inf = std::numeric_limits<long long int>::max()
```

5.23.2.2 negInf

```
const long long int negInf = std::numeric_limits<long long int>::min()
```

5.24 Bellman_Ford.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef BELLMAN_FORD_HPP
#define BELLMAN_FORD_HPP
#include<vector>
#include<utility>
#include<limits>
#include<stddef.h>

struct Edge{
    size_t from, to;
    long long int weight;
};

const long long int inf = std::numeric_limits<long long int>::max();
const long long int negInf = std::numeric_limits<long long int>::min();


/*
My implementation of the Bellman-Ford algorithm. Calculates the distance from startNode to all other nodes in the graph.
Returns a vector of pairs where the first item of each element i is the distance to the node i and the second item is the node you go to i from.
If there is no way to reach a node i then v[i].first is set to inf.
If node i is part of a negative cycle then v[i].first is set to negInf.
The start node's parent is set to inf unless it is part of a negative cycle.
Takes as input the amount of nodes in the graph, which node to compute the distance from, and a vector contaning all Edges in the graph.
Time complexity: O(n*m) where n is the amount of nodes in the graph and m is the amount of edges.
*/
```

```
std::vector<std::pair<long long int, size_t>> Bellman_Ford(const size_t nodes, const size_t startNode, const std::vector<Edge>& edges){
    std::vector<std::pair<long long int, size_t>> dist_pred(nodes, {inf, inf});

    dist_pred[startNode].first = 0;

    //Construct solution
    for(size_t i = 0; i < nodes-1; i++){
        bool change = false;
        for(Edge e : edges){
            if(dist_pred[e.from].first == inf) continue; //Avoid overflow error
            if(dist_pred[e.from].first + e.weight < dist_pred[e.to].first){
                dist_pred[e.to].first = dist_pred[e.from].first + e.weight;
                dist_pred[e.to].second = e.from;
                change = true;
            }
        }
        if(!change) return dist_pred;
    }


    //Now do the same thing again. Every single node that still has a shorter path to it is part of a negative cycle.
    for(size_t i = 0; i < nodes-1; i++){
        bool change = false;
        for(Edge e : edges){
            if(dist_pred[e.from].first == inf) continue; //Avoid overflow error
            if((dist_pred[e.from].first + e.weight < dist_pred[e.to].first) || (dist_pred[e.from].first == negInf && dist_pred[e.to].first != negInf)){
                dist_pred[e.to].first = negInf;
                dist_pred[e.to].second = e.from;
                change = true;
            }
        }
        if(!change) return dist_pred;
    }

    return dist_pred;
}


#endif
```

5.25 Graphs/Dijkstra/shortest_path.hpp File Reference

```
#include <vector>
#include <utility>
#include <queue>
#include <limits>
#include <stddef.h>
```

Macros

- #define node first
- #define parent first
- #define distance second
- #define weight second

Typedefs

- typedef unsigned long long int ull

Functions

- std::vector< std::pair< size_t, ull > > dijkstra_shortest_path (const std::vector< std::vector< std::pair< size_t, ull > > > &adjaceny_list, size_t startNode)

5.25.1 Macro Definition Documentation

5.25.1.1 distance

```
#define distance second
```

5.25.1.2 node

```
#define node first
```

5.25.1.3 parent

```
#define parent first
```

5.25.1.4 weight

```
#define weight second
```

5.25.2 Typedef Documentation

5.25.2.1 ull

```
typedef unsigned long long int ull
```

5.25.3 Function Documentation

5.25.3.1 dijkstra_shortest_path()

```
std::vector< std::pair< size_t, ull > > dijkstra_shortest_path (
            const std::vector< std::vector< std::pair< size_t, ull > > > & adjaceny_list,
            size_t startNode )
```

5.26 shortest_path.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef SHORTEST_PATH_DIJKSTRA_HPP
#define SHORTEST_PATH_DIJKSTRA_HPP
#include<vector>
#include<utility>
#include<queue>
#include<limits>
#include<stddef.h>


/*
My implementation of the dijkstra algorithm.
Returns a vector of pairs where the second item of each element i is the distance to the node i and the first item is the node you go to i from.
If there is no way to reach a node i then v[i].first and v[i].second is set to the max value of size_t and unsigned long long int respectively.
The start node's parent is set to the maximum value of size_t
Takes as input the graph represented in adjacency list form and the node we start from.
Time complexity: O(m * log(n)) where n is the amount of nodes in the graph and m is the amount of edges.
*/

typedef unsigned long long int ull;
#define node first
#define parent first
#define distance second
#define weight second

std::vector<std::pair<size_t, ull» dijkstra_shortest_path(const std::vector<std::vector<std::pair<size_t, ull»>& adjaceny_list, size_t startNode){
    struct Compare{
        bool operator() (std::pair<size_t, ull> a, std::pair<size_t, ull> b){
            return a.distance > b.distance;
        }
    };

    std::vector<std::pair<size_t, ull» parent_dist_pair(adjaceny_list.size(), {std::numeric_limits<size_t>::max(), std::numeric_limits<ull>::max()});
    std::priority_queue<std::pair<size_t, ull>, std::vector<std::pair<size_t, ull», Compare> pq;

    pq.push({startNode, 0});
    parent_dist_pair[startNode].distance = 0;

    while(!pq.empty()){
        std::pair<size_t, ull> p = pq.top();
        pq.pop();

        if(p.distance != parent_dist_pair[p.node].distance)
            continue;

        for(auto edge : adjaceny_list[p.node]){
            if(parent_dist_pair[edge.node].distance > p.distance + edge.weight) {
                pq.push({edge.node, p.distance + edge.weight});
                parent_dist_pair[edge.node].distance = p.distance + edge.weight;
                parent_dist_pair[edge.node].parent = p.node;
            }
        }
    }

    return parent_dist_pair;
}

#endif
```

5.27 Graphs/Dijkstra_Time_Table/shortest_path_time_table.hpp File Reference

```
#include <vector>
#include <utility>
#include <queue>
```

```
#include <limits>
#include <iostream>
#include <stddef.h>
```

Classes

- struct Edge

Macros

- #define node first
- #define parent first
- #define distance second
- #define weight second

Typedefs

- typedef unsigned long long int ull

Functions

- std::vector< std::pair< size_t, unsigned long long int > > dijkstra_shortest_path_time_table (const std↩ ::vector< std::vector< Edge > > &adjaceny_list, size_t startNode)

5.27.1 Macro Definition Documentation

5.27.1.1 distance

```
#define distance second
```

5.27.1.2 node

```
#define node first
```

5.27.1.3 parent

```
#define parent first
```

5.27.1.4 weight

```
#define weight second
```

5.27.2 Typedef Documentation

5.27.2.1 ull

```
typedef unsigned long long int ull
```

5.27.3 Function Documentation

5.27.3.1 dijkstra_shortest_path_time_table()

```
std::vector< std::pair< size_t, unsigned long long int > > dijkstra_shortest_path_time_table
(
            const std::vector< std::vector< Edge > > & adjaceny_list,
            size_t startNode )
```

5.28 shortest_path_time_table.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef SHORTEST_PATH_TIME_TABLE_DIJKSTRA_HPP
#define SHORTEST_PATH_TIME_TABLE_DIJKSTRA_HPP
#include<vector>
#include<utility>
#include<queue>
#include<limits>
#include<iostream>
#include<stddef.h>


struct Edge
{
    size_t destination;
    unsigned long long int travelTime;
    unsigned long long int firstDeparture;
    unsigned long long int repeatDepartures;
};


/*
My implementation of the dijkstra algorithm but were certain edges may only be used on certain times.
Returns a vector of pairs where the second item of each element i is the distance to the node i and the first item is the node you go to i from.
If there is no way to reach a node i then v[i].first and v[i].second is set to the max size of size_t and unsigned long long int respectively.
The start node's parent is set to the maximum value of size_t
Takes as input the graph represented in adjacency list form and the node we start from.
Time complexity: O(m * log(n)) where n is the amount of nodes in the graph and m is the amount of edges.
*/

typedef unsigned long long int ull;
#define node first
#define parent first
#define distance second
#define weight second

std::vector<std::pair<size_t, unsigned long long int» dijkstra_shortest_path_time_table(const std::vector<std::vector<Edge»& adjaceny_list, size_t startNode){
    struct Compare{
        bool operator() (std::pair<size_t, ull> a, std::pair<size_t, ull> b){
            return a.distance > b.distance;
        }
    };

    std::vector<std::pair<size_t, ull» parent_dist_pair(adjaceny_list.size(), {std::numeric_limits<size_t>::max(), std::numeric_limits<ull>::max()});
```

```
    std::priority_queue<std::pair<size_t, ull>, std::vector<std::pair<size_t, ull>>, Compare> pq;

    pq.push({startNode, 0});
    parent_dist_pair[startNode].distance = 0;

    while(!pq.empty()){
        std::pair<size_t, ull> p = pq.top();
        pq.pop();

        if(p.distance != parent_dist_pair[p.node].distance)
            continue;

        for(auto edge : adjaceny_list[p.node]){
            if(edge.repeatDepartures == 0 && p.distance > edge.firstDeparture) continue;
            long long int totalTime = (p.distance > edge.firstDeparture ? p.distance + edge.travelTime + (edge.firstDeparture + edge.repeatDepartures - (p.distance % edge.repeatDepartures))%edge.repeatDepartures : edge.firstDeparture + edge.travelTime);

            if(parent_dist_pair[edge.destination].distance > totalTime) {
                pq.push({edge.destination, totalTime});
                parent_dist_pair[edge.destination].distance = totalTime;
                parent_dist_pair[edge.destination].parent = p.node;
            }
        }
    }

    return parent_dist_pair;
}

#endif
```

5.29 Graphs/Eulerian_Path/Eulerian_Path.hpp File Reference

```
#include <vector>
#include <set>
#include <unordered_set>
#include <stddef.h>
#include <unordered_map>
#include <algorithm>
#include <stack>
#include <iostream>
```

Functions

- std::vector< size_t > Eulerian_Path_Undirected (size_t nodes, const std::vector< std::vector< size_t > > &neighbours)
- std::vector< size_t > Eulerian_Path_Directed (size_t nodes, const std::vector< std::vector< size_t > > &neighbours)

5.29.1 Function Documentation

5.29.1.1 Eulerian_Path_Directed()

```
std::vector< size_t > Eulerian_Path_Directed (
            size_t nodes,
            const std::vector< std::vector< size_t > > & neighbours )
```

5.29.1.2 Eulerian_Path_Undirected()

```
std::vector< size_t > Eulerian_Path_Undirected (
            size_t nodes,
            const std::vector< std::vector< size_t > > & neighbours )
```

5.30 Eulerian_Path.hpp

Go to the documentation of this file.

```
/*
    AUTHOR: Oliver Lindgren
*/

#ifndef EULERIAN_PATH
#define EULERIAN_PATH

#include<vector>
#include<set>
#include<unordered_set>
#include<stddef.h>
#include<unordered_map>
#include<algorithm>
#include<stack>
#include<iostream>

/*
Calculates the Eulerian path in an udirected graph. Takes as input the number of nodes, as well as the edges of the graph in adjaceny list form.
Returns a vector of nodes, indicating the order to visit the nodes in.
If no Eulerian path exists then the empty vector is returned.
Time complexity: O(m) where m is the number of edges in the graph
*/
std::vector<size_t> Eulerian_Path_Undirected(size_t nodes, const std::vector<std::vector<size_t»& neighbours);

/*
Calculates the Eulerian path in a directed graph. Takes as input the number of nodes, as well as the edges of the graph in adjaceny list form.
Returns a vector of nodes, indicating the order to visit the nodes in.
If no Eulerian path exists then the empty vector is returned.
Time complexity: O(m) where m is the number of edges in the graph
*/
std::vector<size_t> Eulerian_Path_Directed(size_t nodes, const std::vector<std::vector<size_t»& neighbours);

std::vector<size_t> Eulerian_Path_Undirected(size_t nodes, const std::vector<std::vector<size_t»& neighbours){
    std::vector<std::unordered_map<size_t, unsigned long long int» adjacent(nodes);

    std::vector<size_t> path;

    size_t edges = 0;
    size_t startNode = 0;
    size_t uneven = 0;

    std::vector<size_t> degrees(nodes);

    for(size_t i = 0; i < nodes; i++){
        for(size_t j : neighbours[i]){
            startNode = i;
            edges++;
            adjacent[i][j]++;
            degrees[i]++;
            degrees[j]++;
        }
    }
    edges /= 2;

    for(size_t i = 0; i < nodes; i++){
```

```
        degrees[i] /= 2;

        if(degrees[i] % 2 != 0){
            startNode = i;
            uneven++;
        }
    }

    if(uneven != 0 && uneven != 2) return {};

    std::stack<size_t> myStack;

    myStack.push(startNode);

    while(!myStack.empty()){
        size_t v = myStack.top();

        if(degrees[v] == 0){
            path.push_back(v);
            myStack.pop();
        }
        else{
            size_t next = adjacent[v].begin()->first;
            adjacent[v][next]--;
            adjacent[next][v]--;
            degrees[v]--;
            degrees[next]--;
            myStack.push(next);
            if(adjacent[v][next] == 0){
                adjacent[v].erase(next);
                adjacent[next].erase(v);
            }
        }
    }
    if(path.size() != edges + 1) return {};
    return path;
}


std::vector<size_t> Eulerian_Path_Directed(size_t nodes, const std::vector<std::vector<size_t»& neighbours){
    std::vector<std::unordered_map<size_t, unsigned long long int» adjacent(nodes);
    std::vector<size_t> path;

    size_t startNode = 0;
    size_t uneven = 0;
    size_t edges = 0;

    std::vector<long long int> degrees(nodes);

    for(size_t i = 0; i < nodes; i++){
        for(size_t j : neighbours[i]){
            startNode = i;
            edges++;
            adjacent[i][j]++;
            degrees[i]++;
            degrees[j]--;
        }
    }

    for(size_t i = 0; i < nodes; i++){
        if(degrees[i] != 0){
            if(degrees[i] == 1)
                startNode = i;
            uneven++;
        }
    }

    if(uneven != 0 && uneven != 2) return {};

    std::stack<size_t> myStack;

    myStack.push(startNode);

    while(!myStack.empty()){
        size_t v = myStack.top();

        if(adjacent[v].empty()){
            path.push_back(v);
            myStack.pop();
        }
        else{
            size_t next = adjacent[v].begin()->first;
            adjacent[v][next]--;

            myStack.push(next);
            if(adjacent[v][next] == 0){
```

```
                adjacent[v].erase(next);
            }
        }
    }

    if(path.size() != edges+1) return {};
    std::reverse(path.begin(), path.end());
    return path;
}

#endif
```

5.31 Graphs/Floyd_Warhsall/floyd_warshall.hpp File Reference

```
#include <vector>
#include <algorithm>
#include <limits>
#include <iostream>
```

Functions

- std::vector< std::vector< long long int > > Floyd_Warshall (std::vector< std::vector< long long int > > distance_matrix)

5.31.1 Function Documentation

5.31.1.1 Floyd_Warshall()

```
std::vector< std::vector< long long int > > Floyd_Warshall (
        std::vector< std::vector< long long int > > distance_matrix )
```

5.32 floyd_warshall.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef FLOYD_WARSHALL_HPP
#define FLOYD_WARSHALL_HPP
#include<vector>
#include<algorithm>
#include<limits>
#include<iostream>


/*
My implementation of the Floyd_Warshall algorithm.
Takes as input the adjacency matrix m of the graph.
If there is no edge between two nodes from u to v then m[u][v] must be set to the maximum value of a long long int divided by 2.
For every node i m[i][i] must be set to 0.
For all nodes u, v with an edge between them from u to w with weight w, m[u][v] must be set to w.
If n is the amount of nodes in the graph then it returns a n*n matrix m where m[u][v] is the distance from u to v.
If it is not possible to reach v from u then m[u][v] is set to the maximum value of a long long int divided by 2
If it is possible to reach v from u taking a path with arbitrarily small value then m[u][v] is set to -(the maximum value of a long long int divided by 2)
Time complexity: O(n^3)
*/
```

```
std::vector<std::vector<long long int» Floyd_Warshall(std::vector<std::vector<long long int» distance_matrix){
    const long long int inf = std::numeric_limits<long long int>::max() / 2;
    size_t n = distance_matrix.size();

    for(size_t k = 0; k < n; k++){
        for(size_t i = 0; i < n; i++){
            for(size_t j = 0; j < n; j++){
                if(distance_matrix[i][k] == inf || distance_matrix[k][j] == inf) continue;
                distance_matrix[i][j] = std::min(distance_matrix[i][j], distance_matrix[i][k] + distance_matrix[k][j]);
            }
        }
    }


    for(size_t i = 0; i < n; i++){
        for(size_t j = 0; j < n; j++){
            for(size_t k = 0; k < n; k++){
                if( distance_matrix[i][k] != inf &&
                    distance_matrix[k][j] != inf &&
                    distance_matrix[k][k] < 0){
                        distance_matrix[i][j] = -inf;
                }
            }
        }
    }

    return distance_matrix;
}

#endif
```

5.33 Graphs/Maximum_Flow/Maximum_Flow.hpp File Reference

```
#include <vector>
#include <stddef.h>
#include <queue>
#include <optional>
```

Classes

- struct Edge

Functions

- bool dinic_bfs (const size_t source, const size_t sink, const std::vector< std::vector< Edge > > &edges, std::vector< long long int > &level)
- long long int dinic_dfs (const size_t at, const size_t sink, std::vector< std::vector< Edge > > &edges, const std::vector< long long int > &level, std::vector< size_t > &next, long long int mini)
- std::pair< long long int, std::vector< Edge > > dinic_max_flow (const size_t nodes, const size_t source, const size_t sink, std::vector< std::vector< Edge > > edges)

5.33.1 Function Documentation

5.33.1.1 dinic_bfs()

```
bool dinic_bfs (
            const size_t source,
            const size_t sink,
            const std::vector< std::vector< Edge > > & edges,
            std::vector< long long int > & level )
```

5.33.1.2 dinic_dfs()

```
long long int dinic_dfs (
            const size_t at,
            const size_t sink,
            std::vector< std::vector< Edge > > & edges,
            const std::vector< long long int > & level,
            std::vector< size_t > & next,
            long long int mini )
```

5.33.1.3 dinic_max_flow()

```
std::pair< long long int, std::vector< Edge > > dinic_max_flow (
            const size_t nodes,
            const size_t source,
            const size_t sink,
            std::vector< std::vector< Edge > > edges )
```

5.34 Maximum_Flow.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef MAXIMUM_FLOW_HPP
#define MAXIMUM_FLOW_HPP

#include<vector>
#include<stddef.h>
#include<queue>
#include<optional>

struct Edge {
    size_t from; //Vilken nod går denna kanten från?
    size_t to; //Vilken nod går denna kanten till?
    long long int capacity; //Hur mycket capacitet finns kvar?
    long long int flow = 0; //Hur mycket flöde är det just nu?
    long long int reverse = -1; //Var finns reverse-flow kanten?
};

//Beräkna nivån för varje nod. D.v.s. hur långt ifrån källan den ligger. Om en nod inte kan nås från källan då får den värdet -1
bool dinic_bfs(const size_t source, const size_t sink, const std::vector<std::vector<Edge»& edges, std::vector<long long int>& level);

//Hitta vägar som expanderar flödet med en DFS.
long long int dinic_dfs(const size_t at, const size_t sink, std::vector<std::vector<Edge»& edges, const std::vector<long long int>& level, std::vector<size_t>& next, long long int mini);


/*
My implementation of dinitz algorithm to calculate the maximum flow in a graph.
Takes as input the number of nodes in the graph, what node is the source, what node is the sink, and the edges of the graph in adjacency list representation.
Returns a pair, where the first element is the size of the maximum possible flow for the graph. The second element is a vector containing all edges that there is flow through.
Time complexity: O(m*n^2) where n is the number of nodes in the graph, and m is the number of edges.
*/
std::pair<long long int, std::vector<Edge» dinic_max_flow(const size_t nodes, const size_t source, const size_t sink, std::vector<std::vector<Edge» edges);
```

```
bool dinic_bfs(const size_t source, const size_t sink, const std::vector<std::vector<Edge»& edges, std::vector<long long int>& level){
    for (size_t i = 0; i < level.size(); i++) {
        level[i] = -1;
    }

    level[source] = 0;

    std::queue<size_t> q;
    q.push(source);
    size_t at;
    while (!q.empty()) {
        at = q.front();
        q.pop();

        for (size_t i = 0; i < edges[at].size(); i++) {
            if (level[edges[at][i].to] < 0 && edges[at][i].flow < edges[at][i].capacity) {
                level[edges[at][i].to] = level[at] + 1;
                q.push(edges[at][i].to);
            }
        }
    }

    return level[sink] > 0; //Om level[sink] = -1 då finns det ingen väg från källan till sänkan och det finns inget sätt att skicka mer flöde.
}

long long int dinic_dfs(const size_t at, const size_t sink, std::vector<std::vector<Edge»& edges, const std::vector<long long int>& level, std::vector<size_t>& next, long long int mini){
    if (at == sink) return mini;

    //Next sparas mellan olika anrop av dfs. D.v.s vi börjar där vi vet att det kan finnas utrymme att expandera.
    for (; next[at] < edges[at].size(); next[at]++) {
        Edge& e = edges[at][next[at]];

        //Om noden vi tittar på har en högre level och kanten har kapacitet.
        if (level[e.to] == level[at] + 1 && e.flow < e.capacity) {

            //T1 är den minsta kapaciteten hos alla kanter som går till sänkan. D.v.s. maximala möjliga utvidgninen.
            long long int t1 = dinic_dfs(e.to, sink, edges, level, next, std::min(mini, e.capacity - e.flow));
            if (t1) {
                e.flow += t1;
                edges[e.to][e.reverse].flow -= t1;
                return t1;
            }
        }
    }
    return 0;
}

std::pair<long long int, std::vector<Edge» dinic_max_flow(const size_t nodes, const size_t source, const size_t sink, std::vector<std::vector<Edge» edges){
    long long int flowIncrease;
    long long int total = 0;

    //Add reverse flow edges
    for(size_t node = 0; node < nodes; node++){
        for(size_t i = 0; i < edges[node].size(); i++){
```

```
            Edge& e = edges[node][i];
            if (e.reverse == -1){
                e.reverse = edges[e.to].size();
                Edge reverse = {e.to, e.from, 0, 0, i};
                edges[e.to].push_back(reverse);
            }
            else{
                break;
            }
        }
    }

    std::vector<long long int> level(nodes);
    std::vector<size_t> next(2*nodes);
    while (dinic_bfs(source, sink, edges, level)) {
        for (size_t i = 0; i < nodes; i++) {
            next[i] = 0;
        }
        do {
            flowIncrease = dinic_dfs(source, sink, edges, level, next, (1 « 30));
            total += flowIncrease;

        } while (flowIncrease); //Medans det finns flöde att expandera med, gör det.
    }

    std::vector<Edge> flowEdges;
    for (size_t i = 0; i < nodes; i++) {
        for (size_t j = 0; j < edges[i].size(); j++) {
            if (edges[i][j].flow > 0) flowEdges.push_back(edges[i][j]);
        }
    }

    return {total, flowEdges};
}




#endif
```

5.35 Graphs/Minimum_Cut/Minimum_Cut.hpp File Reference

```
#include "Graphs\Maximum_Flow\Maximum_Flow.hpp"
```

Functions

- std::vector< size_t > minimum_cut (const size_t nodes, const size_t s, const size_t t, std::vector< std↩ ::vector< Edge > > edges)

5.35.1 Function Documentation

5.35.1.1 minimum_cut()

```
std::vector< size_t > minimum_cut (
            const size_t nodes,
            const size_t s,
            const size_t t,
            std::vector< std::vector< Edge > > edges )
```

5.36 Minimum_Cut.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef MINIMUM_CUT
#define MINIMUM_CUT

#include "Graphs\Maximum_Flow\Maximum_Flow.hpp"

/*
My implementation of using dinitz algorithm to calculate the minimum cut of a graph.
Takes as input the number of nodes in the graph, the vertice s that should be in the cut, the vertice v which must not be in the cut. And the edges of the graph as adjacency lists
Returns a vector containing all the nodes that are included in the cut.
Time complexity: O(m*n^2) where n is the number of nodes in the graph, and m is the number of edges.
*/

std::vector<size_t> minimum_cut(const size_t nodes, const size_t s, const size_t t, std::vector<std::vector<Edge» edges){
    long long int flowIncrease;
    long long int total = 0;

    //Add reverse flow edges
    for(size_t node = 0; node < nodes; node++){
        for(size_t i = 0; i < edges[node].size(); i++){
            Edge& e = edges[node][i];
            if (e.reverse == -1){
                e.reverse = edges[e.to].size();
                Edge reverse = {e.to, e.from, 0, 0, i};
                edges[e.to].push_back(reverse);
            }
            else{
                break;
            }
        }
    }

    std::vector<long long int> level(nodes);
    std::vector<size_t> next(2*nodes);
    while (dinic_bfs(s, t, edges, level)) {
        for (size_t i = 0; i < nodes; i++) {
            next[i] = 0;
        }
        do {
            flowIncrease = dinic_dfs(s, t, edges, level, next, (1 « 30));
            total += flowIncrease;

        } while (flowIncrease); //Medans det finns flöde att expandera med, gör det.
    }

    //The nodes to be included in the cut are all nodes still reachable from s
    std::vector<size_t> minCutNodes;
    for (size_t i = 0; i < nodes; i++) {
        if(level[i] != -1) minCutNodes.push_back(i);
    }

    return minCutNodes;
}

#endif
```

5.37 Graphs/MST/MST.hpp File Reference

```
#include <vector>
#include <utility>
#include <algorithm>
#include <stddef.h>
#include "Data_Structures\Disjoint_Set\DisjointSet.hpp"
```

Functions

- std::pair< long long int, std::vector< std::pair< size_t, size_t > > > mst (size_t vertices, std::vector< std::pair< long long int, std::pair< size_t, size_t > > > edges)

5.37.1 Function Documentation

5.37.1.1 mst()

```
std::pair< long long int, std::vector< std::pair< size_t, size_t > > > mst (
            size_t vertices,
            std::vector< std::pair< long long int, std::pair< size_t, size_t > > > edges )
```

5.38 MST.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef MINIMIAL_SPANNING_TREE_HPP
#define MINIMIAL_SPANNING_TREE_HPP

#include<vector>
#include<utility>
#include<algorithm>
#include<stddef.h>
#include"Data_Structures\Disjoint_Set\DisjointSet.hpp"

/*
Function used to calculate the minimum spanning tree of a graph.
Takes as input the number of vertices and a vector of edges.
An edge is represented as a pair of 1. the edge weight w, and 2. a pair containing nodes u & v. This means there is an edge between u & v with weight w
Returns a pair containing 1. the size of the tree, and 2. a vector containing pairs what nodes there are edges between.
If there is no minimum spanning tree then the size is returned as -1 and the vector is empty.
Time complexity: O(m * log(n)) where n is the number of nodes and m is the number of edges in the graph.
*/

std::pair<long long int, std::vector<std::pair<size_t, size_t»> mst(size_t vertices, std::vector<std::pair<long long int, std::pair<size_t, size_t»> edges){
    long long int cost = 0;
    std::vector<std::pair<size_t,size_t» included;
    std::sort(edges.begin(), edges.end());
    DisjointSet s(vertices);
    for(auto a : edges) {
        if (!s.query(a.second.first, a.second.second)) {
            s.unionSets(a.second.first, a.second.second);
            included.push_back({a.second.first, a.second.second});
            cost += a.first;
        }
    }

    if(included.size() != vertices-1) return {-1, {}};
    return {cost,included};
}

#endif
```

5.39 Misc/Equation solver plus/solver.hpp File Reference

```
#include <vector>
#include <cmath>
#include <iostream>
#include <assert.h>
```

Functions

- bool is_double_zero (double x)
- std::vector< double > equation_solver (const std::vector< std::vector< double > > &a, const std::vector< double > &b)

5.39.1 Function Documentation

5.39.1.1 equation_solver()

```
std::vector< double > equation_solver (
            const std::vector< std::vector< double > > & a,
            const std::vector< double > & b )
```

5.39.1.2 is_double_zero()

```
bool is_double_zero (
            double x )
```

5.40 solver.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef EQUATION_SOLVER_HPP
#define EQUATION_SOLVER_HPP
#include<vector>
#include<cmath>
#include<iostream>
#include<assert.h>


//avoid silly precision errors
bool is_double_zero(double x){
    if(x < 0) x = -x;
    return x < 0.000000001;
}

/*
Tar en n X n matris a och en vector b av längd n. Löser sedan ekvationssystemet ax = b med hjälp av gauss-elimininering
och returnerar vectorn v av längd n där v[i] = värdet för x_i
Om ett x_i inte kan bestämmas utan har flera möjliga värden då är v[i] = NaN.
Om det linjära ekvationssystemet är inkonsistent då returneras en tom vektor.
*/

std::vector<double> equation_solver(const std::vector<std::vector<double»& a, const std::vector<double>& b){
    std::vector<std::vector<double» m = a;
    std::vector<double> ans;

    for(int row = 0; row < a.size(); row++){
        m[row].push_back(b[row]);
    }

    int rowStart = 0;

    //Triangulering
    for(int col = 0; col < m.size(); col++){
        int index = col;


        for(int row = rowStart; row < m.size(); row++){
            if(std::abs(m[row][col]) > std::abs(m[index][col])) index = row;
```

```
        }

        if(is_double_zero(m[index][col])){
            continue;
        }


        m[rowStart].swap(m[index]);

        for(int col2 = m[0].size()-1; col2 >= col; col2--){
            m[rowStart][col2] /= m[rowStart][col];
        }

        for(int row = rowStart+1; row < m.size(); row++){
            if(is_double_zero(m[rowStart][col])) continue;
            double mult = m[row][col]/m[rowStart][col];

            for(int col2 = col; col2 < m[row].size(); col2++){
                m[row][col2] -= m[rowStart][col2] * mult;
            }
        }

        rowStart++;
    }


    //Bakåtsubstitution
    for(int row1 = m.size()-1; row1 > 0; row1--){
        int col=-1;
        for(int column = 0; column < m.size(); column++){
            if(m[row1][column] == 1){
                col = column;
                break;
            }
        }
        if(col == -1) continue;

        for(int row2 = row1-1; row2 >= 0; row2--){
            if(is_double_zero(m[row1][col])) continue;
            double mult = m[row2][col]/m[row1][col];

            for(int col = 0; col <= m.size(); col++){
                m[row2][col] -= m[row1][col]*mult;
            }

        }
    }

    //Check for inconsistency
    for(int row = 0; row < m.size(); row++){
        bool allZero = true;

        for(int col = 0; col < m.size(); col++){
            if(!is_double_zero(m[row][col])) allZero = false;
        }

        if(allZero && !is_double_zero(m[row][m.size()])) return {};
    }

    //Construct solution
    rowStart = 0;
    for(int col = 0; col < m.size(); col++){
        if(is_double_zero(m[rowStart][col])){
            ans.push_back(std::nan(""));
            continue;
        }

        bool allZero = true;

        for(int col2 = col+1; col2 < m.size(); col2++){
            if(!is_double_zero(m[rowStart][col2])) allZero = false;
        }

        if(allZero == false) ans.push_back(std::nan(""));
        else ans.push_back(m[rowStart][m.size()]);

        rowStart++;
    }
    assert(ans.size() == b.size());
    return ans;

}

#endif
```

5.41 Misc/Interval Cover/cover.hpp File Reference

```
#include <vector>
#include <utility>
#include <algorithm>
```

Functions

- std::vector< size_t > cover (const std::pair< double, double > interval, const std::vector< std::pair< double, double > > &intervals)

5.41.1 Function Documentation

5.41.1.1 cover()

```
std::vector< size_t > cover (
            const std::pair< double, double > interval,
            const std::vector< std::pair< double, double > > & intervals )
```

5.42 cover.hpp

Go to the documentation of this file.

```
/*
    AUTHOR: Oliver Lindgren
*/

#ifndef COVER_HPP
#define COVER_HPP

#include<vector>
#include<utility>
#include<algorithm>

/*
Takes an interval to be covered as a pair of doubles, with the first pair value being the left bound of the interval,
and the second pair value being the right bound of the interval. It also takes a list of intervals that should be used to
try and cover this interval. Returns a vector containing the indices in the intervals vector relating to the intervals that need to be used.
If it is not possible to cover interval then an empty vector is returned.
O(n log(n)) - where n is the size of the intervals vector.

TODO: Template it & convert intervals to be range based.
*/


std::vector<size_t> cover(const std::pair<double, double> interval, const std::vector<std::pair<double, double»& intervals){

    //Tripple will be used later to store intervals (first, second) as well as their index in the originial intervals array.
    struct Tripple{
        double first, second;
        size_t third;
    };

    double l = interval.first, r = interval.second;
    if(l == r){ //If the interval is a single point then deal with that case
        for(size_t i = 0; i < intervals.size(); i++){
            if(intervals[i].first <= l && intervals[i].second >= r) return {i};
        }

        return {};
```

```
    }


    //Make a copy of intervals (where the original index is included also) and sort it based on where the intervals begin. This will be necessary to use the greedy algorithm effectively.
    std::vector<Tripple> intervalsCopy;

    intervalsCopy.reserve(intervals.size());

    for(size_t i = 0; i < intervals.size(); i++){
        intervalsCopy.push_back({intervals[i].first, intervals[i].second, i});
    }

    std::sort(intervalsCopy.begin(), intervalsCopy.end(), [](Tripple& a, Tripple& b){
        return a.first < b.first;
    });

    size_t index = 0;
    size_t bestIndex = 0;
    double reach = l;

    std::vector<size_t> solution;


    while(l < r){
        while(index < intervalsCopy.size() && intervalsCopy[index].first <= l){

            //Find whatever interval that begins before current l that reaches the furthest.
            if(intervalsCopy[index].second > reach){
                bestIndex = intervalsCopy[index].third;
                reach = intervalsCopy[index].second;
            }
            index++;
        }

        //If we couldn't extend the interval we are working on then return empty vector.
        if(reach == l) return {};

        l = reach;
        solution.push_back(bestIndex);
    }

    return solution;
}


#endif
```

5.43 Misc/Knapsack/Knapsack.hpp File Reference

```
#include <vector>
#include <utility>
#include <algorithm>
```

Functions

- std::vector< size_t > knapsack (const int capacity, const std::vector< std::pair< int, int > > &items)

5.43.1 Function Documentation

5.43.1.1 knapsack()

```
std::vector< size_t > knapsack (
            const int capacity,
            const std::vector< std::pair< int, int > > & items )
```

5.44 Knapsack.hpp

Go to the documentation of this file.

```
/*
    AUTHOR: Oliver Lindgren
*/

#ifndef KNAPSACK_HPP
#define KNAPSACK_HPP
#include<vector>
#include<utility>
#include<algorithm>

/*
A function that solves the knapsack problem and returns the indices in the items vector that should be used to obtain the optimal value.
Capacity is the maximum weight that the knapsack can hold. Must be non-negative.
Items is a list of non-negative integer pairs where the first value in each pair is the weight of the item, and the second pair is the value.
O(n*k) - where n is the size of the items vector and k is the capacity

TODO: Change to iterator based implementation, make capacity size_t, template it.
*/

std::vector<size_t> knapsack(const int capacity, const std::vector<std::pair<int, int»& items){
    std::vector<size_t> indices;

    std::vector<std::vector<int» dp(capacity+1);
    for(size_t i = 0; i <= capacity; i++){
        dp[i].resize(items.size()+1);
    }

    //Fill out DP
    for(int weight = capacity; weight >= 0; --weight){
        for(int item = items.size(); item >= 0; --item){
            if(weight == capacity || item == items.size())
                dp[weight][item] = 0;
            else if (weight+items[item].first <= capacity)
                dp[weight][item] = std::max(dp[weight][item + 1], dp[weight+items[item].first][item + 1] + items[item].second);
            else
                dp[weight][item] = dp[weight][item + 1];
        }
    }

    //Reconstruct the solution
    int weight = 0;
    size_t item = 0;
    while(item != items.size() && weight != capacity){
        if (weight+items[item].first <= capacity && dp[weight+items[item].first][item + 1] + items[item].second > dp[weight][item + 1]){
            indices.push_back(item);
            weight += items[item].first;
            item++;
        }
        else{
            item++;
        }
    }

    return indices;
}


#endif
```

5.45 Misc/Longest Increasing Subsequence/lis.hpp File Reference

```
#include <vector>
#include <iterator>
#include <limits>
#include <algorithm>
```

Functions

- std::vector< size_t > lis (const std::vector< long long int > &v)

5.45.1 Function Documentation

5.45.1.1 lis()

```
std::vector< size_t > lis (
            const std::vector< long long int > & v )
```

5.46 lis.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef LONGEST_INCREASING_SUBSEQUENCE_HPP
#define LONGEST_INCREASING_SUBSEQUENCE_HPP
#include<vector>
#include<iterator>
#include<limits>
#include<algorithm>

/*
Takes a vector of integers and returns a vector of the indicies that should be used if you want to construct the longest possible subsequence using those integers.
O(N*log(N))
*/

std::vector<size_t> lis(const std::vector<long long int>& v){
    typedef long long int ll;
    const ll inf = std::numeric_limits<ll>::max();

    std::vector<size_t> ans;
    std::vector<std::pair<ll, size_t>> last(v.size()+1, {inf, inf}); //Last[L].first is the lowest value so far that can end a subsequence of length L. Second är L's index i v
    std::vector<size_t> backTrack(v.size(), inf);
    last[0].first = -last[0].first;

    for(size_t i = 0; i < v.size(); i++){

        //it pekar på första elementet som är större än v[i].
        auto it = std::upper_bound(last.begin(), last.end(), v[i], [](auto value, auto elem){
            return value < elem.first;
        });

        if((it-1)->first < v[i]){ //Kolla så att det går att avsluta en subsequence av längd L-1 med ett värde som är mindre än v[i]
            it->first = v[i];
            it->second = i;
            backTrack[i] = (it-1)->second;
        }
    }

    size_t L;
    for(size_t i = 1; i < last.size(); i++){
        if(last[i].first < inf){
            L = last[i].second;
        }
    }

    //Reconstruct the solution
    ans.reserve(L);
    do{
        ans.push_back(L);
        L = backTrack[L];
    }while(L != inf);

    std::reverse(ans.begin(), ans.end());

    return ans;
}


#endif
```

5.47 Misc/Polymul/FFT.hpp File Reference

```
#include <vector>
#include <complex>
```

Macros

- #define PI 3.14159265359

Typedefs

- typedef long long int ll

Functions

- std::vector< ll > Polymul (std::vector< ll > a, std::vector< ll > b)
- std::vector< ll > Polymul_Quadratic (std::vector< ll > a, std::vector< ll > b)
- std::vector< std::complex< double > > FFT (const std::vector< ll > &P, int n, int start=0, int skip=1)
- std::vector< std::complex< double > > INVERSE_FFT (const std::vector< std::complex< double > > &P, int n, int start=0, int skip=1)

5.47.1 Macro Definition Documentation

5.47.1.1 PI

```
#define PI 3.14159265359
```

5.47.2 Typedef Documentation

5.47.2.1 ll

```
typedef long long int ll
```

5.47.3 Function Documentation

5.47.3.1 FFT()

```
std::vector< std::complex< double > > FFT (
            const std::vector< ll > & P,
            int n,
            int start = 0,
            int skip = 1 )
```

5.47.3.2 INVERSE_FFT()

```
std::vector< std::complex< double > > INVERSE_FFT (
            const std::vector< std::complex< double > > & P,
            int n,
            int start = 0,
            int skip = 1 )
```

5.47.3.3 Polymul()

```
std::vector< ll > Polymul (
            std::vector< ll > a,
            std::vector< ll > b )
```

5.47.3.4 Polymul_Quadratic()

```
std::vector< ll > Polymul_Quadratic (
            std::vector< ll > a,
            std::vector< ll > b )
```

5.48 FFT.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/


#ifndef FFT_HPP
#define FFT_HPP

#define PI 3.14159265359

#include<vector>
#include<complex>

typedef long long int ll;
/*
Takes two polynomials in coefficient list representation, where the coefficients are whole integers,
multiplies them together using the fast fourier transformtation algorithm
and returns the resulting polynomial also in coefficient list representation.
O(N log(N))
*/
std::vector<ll> Polymul(std::vector<ll> a, std::vector<ll> b);

/*
Takes two polynomials in coefficient list representation, where the coefficients are whole integers,
multiplies them together using a naive O(N^2) algorithm
and returns the resulting polynomial also in coefficient list representation.
O(N^2)
*/
std::vector<ll> Polymul_Quadratic(std::vector<ll> a, std::vector<ll> b);

/*
Applies the FFT on the vector P and returns the result. N is the size of the vector P.
Start and skip are used to index the vector and should be left default initialized.
O(N log(N))
*/
std::vector<std::complex<double» FFT(const std::vector<ll>& P, int n, int start = 0, int skip = 1);

/*
Applies the inverse FFT on the vector P and returns the result. N is the size of the vector P.
Start and skip are used to index the vector and should be left default initialized.
O(N log(N))
*/
```

```
std::vector<std::complex<double>> INVERSE_FFT(const std::vector<std::complex<double>>& P, int n, int start = 0, int skip = 1);






























std::vector<ll> Polymul(std::vector<ll> a, std::vector<ll> b){
    int n=1;
    int org = a.size() + b.size() - 1;

    while(n < org) n*=2;

    while(a.size() < n) a.push_back(0);
    while(b.size() < n) b.push_back(0);

    std::vector<std::complex<double>> a_value_rep = FFT(a, n);
    std::vector<std::complex<double>> b_value_rep = FFT(b, n);
    std::vector<std::complex<double>> merged;
    std::vector<std::complex<double>> reversed;

    merged.reserve(n);

    for(int i = 0; i < n; i++){
        merged.push_back(a_value_rep[i] * b_value_rep[i]);
    }

    reversed = INVERSE_FFT(merged, n);

    std::vector<ll> ans;
    ans.reserve(org);

    for(int i = 0; i < org; i++){
        ans.push_back(round(reversed[i].real()/(double)n));
    }

    return ans;
}


std::vector<std::complex<double>> FFT(const std::vector<ll>& p, int n, int start, int skip){
    using namespace std::complex_literals;

    if(p.size() == skip){
        return {{(double)p[start], 0.0}};
    } //Ett polynom av grad 0 har alltid värdet av koefficienten. FIIIIIIIX



    std::vector<std::complex<double>> y_even, y_odd;
    y_even = FFT(p, n/2, start, skip*2);
    y_odd = FFT(p, n/2, start+skip, skip*2);

    std::vector<std::complex<double>> y_merged(n);
    for(int i = 0; i < n/2; i++){
        std::complex<double> compute = std::exp((2.0*PI*1i*(double)i)/(double)n) * y_odd[i];
        y_merged[i] = y_even[i] + compute;
        y_merged[i+n/2] = y_even[i] - compute;
    }
    return y_merged;
}

```

```

std::vector<std::complex<double>> INVERSE_FFT(const std::vector<std::complex<double>>& p, int n, int
start, int skip){
    using namespace std::complex_literals;

    if(p.size() == skip){
        return {p[start]};
    } //Ett polynom av grad 0 har alltid värdet av koefficienten. FIIIIIIIX

    std::vector<std::complex<double>> y_even, y_odd;
    y_even = INVERSE_FFT(p, n/2, start, skip*2);
    y_odd = INVERSE_FFT(p, n/2, start+skip, skip*2);

    std::vector<std::complex<double>> y_merged(n);
    for(int i = 0; i < n/2; i++){
        std::complex<double> compute = std::exp((-2.0*PI*1i*(double)i)/(double)n) * y_odd[i];
        y_merged[i] = y_even[i] + compute;
        y_merged[i+n/2] = y_even[i] - compute;
    }
    return y_merged;
}

std::vector<ll> Polymul_Quadratic(std::vector<ll> a, std::vector<ll> b){
    std::vector<ll> res(a.size() + b.size() - 1);

    for(int i = 0; i < a.size(); i++){
        for(int j = 0; j < b.size(); j++){
            res[i+j] += a[i]*b[j];
        }
    }

    return res;
}

#endif
```

5.49 Prime_Numbers/Prime_Factorisation/Factorize.hpp File Reference

```
#include <cmath>
#include <vector>
#include "Prime_Numbers\Prime_Sieve\Sieve.hpp"
```

Typedefs

- typedef long long int ll
- typedef unsigned long long int ull
- typedef __uint128_t u128

Functions

- ull modMult (const ull a, const ull b, const ull m)
- ull powMod (ull base, ull exp, ull m)
- bool Miller_test (ull n, ull a, ull d, int s)
- bool is_Probably_Prime (ull n, int k=5)
- ll gcd (ll a, ll b)
- ull f (ull x, ull c, ull m)
- ull Pollard_Rho (ull n, ull c=1, ull x0=2)
- vector< pair< ull, ull > > prime_with_mult (ull n)
- vector< ull > fast_prime (ull n)
- void nested (vector< pair< ull, ull > > &v, const vector< pair< ull, ull > > &org, int depth, vector< ull > &ans)
- vector< ull > fast_factors (ull n)

Variables

- vector< size_t > primes

5.49.1 Typedef Documentation

5.49.1.1 ll

```
typedef long long int ll
```

5.49.1.2 u128

```
typedef __uint128_t u128
```

5.49.1.3 ull

```
typedef unsigned long long int ull
```

5.49.2 Function Documentation

5.49.2.1 f()

```
ull f (
            ull x,
            ull c,
            ull m )  [inline]
```

5.49.2.2 fast_factors()

```
vector< ull > fast_factors (
            ull n )
```

5.49.2.3 fast_prime()

```
vector< ull > fast_prime (
            ull n )
```

5.49.2.4 gcd()

```
ll gcd (
            ll a,
            ll b )
```

5.49.2.5 is_Probably_Prime()

```
bool is_Probably_Prime (
            ull n,
            int k = 5 )
```

5.49.2.6 Miller_test()

```
bool Miller_test (
            ull n,
            ull a,
            ull d,
            int s )
```

5.49.2.7 modMult()

```
ull modMult (
            const ull a,
            const ull b,
            const ull m )  [inline]
```

5.49.2.8 nested()

```
void nested (
            vector< pair< ull, ull > > & v,
            const vector< pair< ull, ull > > & org,
            int depth,
            vector< ull > & ans )
```

5.49.2.9 Pollard_Rho()

```
ull Pollard_Rho (
            ull n,
            ull c = 1,
            ull x0 = 2 )
```

5.49.2.10 powMod()

```
ull powMod (
            ull base,
            ull exp,
            ull m )
```

5.49.2.11 prime_with_mult()

```
vector< pair< ull, ull > > prime_with_mult (
            ull n )
```

5.49.3 Variable Documentation

5.49.3.1 primes

```
vector<size_t> primes
```

5.50 Factorize.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/


#include<cmath>
#include<vector>

#include"Prime_Numbers\Prime_Sieve\Sieve.hpp"

using namespace std;

typedef long long int ll;
typedef unsigned long long int ull;
typedef __uint128_t u128;

vector<size_t> primes;

inline ull modMult(const ull a, const ull b, const ull m){
    return (u128)a * b % m;
}

ull powMod(ull base, ull exp, ull m) {
    ull result = 1;
    base %= m;
    while (exp) {
        if (exp % 2 == 1)
            result = (u128)result * base % m;
        base = (u128)base * base % m;
        exp /= 2;
    }
    return result;
}

bool Miller_test(ull n, ull a, ull d, int s) {
    ull x = powMod(a, d, n);
    if (x == 1 || x == n - 1)
        return false;
    for (int r = 1; r < s; r++) {
        x = modMult(x, x, n);
        if (x == n - 1)
            return false;
    }
    return true;
};

bool is_Probably_Prime(ull n, int k=5) { // returns true if n is probably prime, else returns false.
    if (n == 1 || n == 4)
        return false;
    if (n <= 5) return true;

    int s = 0;
    ull d = n - 1;
    while (d % 2 == 0) {
        d /= 2;
        s++;
    }

    for (int i = 0; i < k; i++) {
        int a = 2 + rand() % (n - 3);
        if (Miller_test(n, a, d, s))
            return false;
    }
    return true;
}

ll gcd(ll a, ll b){
    a = std::abs(a);
```

```
    b = std::abs(b);
    if(a > b){
        std::swap(a,b);
    }

    if (a == 0)
        return b;
    return gcd(b % a, a);
}

inline ull f(ull x, ull c, ull m){
    return (modMult(x, x, m) + c) % m;
}

ull Pollard_Rho(ull n, ull c = 1, ull x0 = 2){
    ll hare = x0;
    ll tortoise = x0;
    ull p = 1;


    ull t = 0;
    while(p == 1){
        t++;
        tortoise = f(tortoise, c, n);
        hare = f(hare, c, n);
        hare = f(hare, c, n);

        p = gcd(abs(tortoise-hare), n);
    }

    if(p == n) return Pollard_Rho(n, c+1, x0);
    else return p;
}

//Almost identical to fast_prime except it keeps track of how many times a prime appears
vector<pair<ull,ull» prime_with_mult(ull n){
    vector<pair<ull,ull» ans;

    for(auto x : primes){
        if(n % x == 0) ans.push_back({x, 0});
        while(n % x == 0){
            n /= x;
            ans.rbegin()->second++;
        }

        if(x*x > n) break;
    }

    if(n == 1) return ans;

    if(!is_Probably_Prime(n, 10)){
        ull p = Pollard_Rho(n);
        if(p*p == n){
            ans.push_back({p,2});
        }
        else{
            ans.push_back({p, 1});
            ans.push_back({n/p, 1});
        }
    }
    else{
        ans.push_back({n, 1});
    }
    return ans;
}

vector<ull> fast_prime(ull n){
    vector<ull> ans;

    for(auto x : primes){
        if(n % x == 0) ans.push_back(x);
        while(n % x == 0) n /= x;

        if(x*x > n) break;
    }

    if(n == 1) return ans;

    if(!is_Probably_Prime(n, 10)){
        ull p = Pollard_Rho(n);

        if(p*p == n){
            ans.push_back(p);
        }
        else{
            ans.push_back(p);
            ans.push_back(n/p);
```

```
        }
    }
    else{
        ans.push_back(n);
    }
    return ans;
}

void nested(vector<pair<ull,ull»& v, const vector<pair<ull,ull»& org, int depth, vector<ull>& ans){
    if(depth == v.size()){
        ull d = 1;
        for(int i = 0; i < v.size(); i++){
            d *= pow(v[i].first, v[i].second);
        }
        ans.push_back(d);
        return;
    }

    while(v[depth].second >= 0){
        nested(v, org, depth+1, ans);

        if(v[depth].second == 0)break;
        v[depth].second--;
    }
    v[depth].second = org[depth].second;
    return;
}


vector<ull> fast_factors(ull n){
    Prime_Sieve sieve(4e4 + 7e3); //Ty 14/3 ~= 4.7
    primes = sieve.get_Primes();


    auto v = prime_with_mult(n);
    auto org = v;
    vector<ull> ans;

    nested(v, org, 0, ans);
    return ans;
}
```

5.51 Prime_Numbers/Prime_Sieve/Sieve.hpp File Reference

```
#include <vector>
#include <stddef.h>
#include <iostream>
```

Classes

- class Prime_Sieve

5.52 Sieve.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef PRIME_SIEVE_HPP
#define PRIME_SIEVE_HPP

#include<vector>
#include<stddef.h>
#include<iostream>

/*
    Class to compute all primes up to a number N. it provides the following functions:

```

```
    Prime_Sieve(N) - the constructor takes a number N as an argument and computes which numbers up to N that are prime.
    get_Number_Of_Primes() - returns the number of primes that are smaller than or equal to N.
    get_Primes() returns a sorted vector of all numbers less than N that are prime.
    is_Prime(x) - returns true if x is prime. Returns false otherwise.
*/

class Prime_Sieve{
    std::vector<bool> is_composite;
    std::vector<size_t> primes;

    public:
    const std::vector<size_t>& get_Primes();
    Prime_Sieve(size_t n);
    size_t get_Number_Of_Primes();
    bool is_Prime(size_t i);
};

Prime_Sieve::Prime_Sieve(size_t n) : is_composite(n+1){
    n++;
    is_composite[0] = true;              //0 is not a composite number, however, it is not prime either.
    if(n > 1) is_composite[1] = true;    //The same applies to the number 1.
    if(n > 2) primes.push_back(2);


    size_t x = 4;
    while(x < n){
        is_composite[x] = true;
        x+=2;
    }


    for(size_t i = 3; i < n; i+=2){
        if(is_composite[i]) continue;

        primes.push_back(i);

        size_t mult = i*i;
        size_t add = 2*i;
        while(mult < n){
            is_composite[mult] = true;
            mult += add;
        }

    }

}

bool Prime_Sieve::is_Prime(size_t i){
    return !is_composite[i];
}
```

```

size_t Prime_Sieve::get_Number_Of_Primes(){
    return primes.size();
}

const std::vector<size_t>& Prime_Sieve::get_Primes(){
    return primes;
}

#endif
```

5.53 README.md File Reference

5.54 Strings/Aho-corasick/Aho-Corasick.hpp File Reference

```
#include <vector>
#include <string>
#include <queue>
#include <iostream>
```

Classes

- class TrieAutomaton

5.55 Aho-Corasick.hpp

Go to the documentation of this file.

```
/*
Author: Oliver Lindgren
*/

#ifndef AHO_CORASICK_HPP
#define AHO_CORASICK_HPP
#include<vector>
#include<string>
#include<queue>
#include<iostream>

/*
A class that implements the Aho-Corasick string matching algorithm.
It provides three functions of interest:

add_string(s) adds a string pattern to be matched against.
construct_automaton() constructs the automaton that will be used in the string matching. Call this once all strings you want to match against have been added.
search(s) searches through the string s. Returns a vector V of vectors containing the indicies in s where a pattern was sucessfully matched. V[i] contains the positions of where the ith string added match.
*/

class TrieAutomaton{
struct Node{
    std::vector<int> terminal;
    int depth;
    Node* suffixNode = nullptr;
    Node* terminalChain;
    char nodeChar = 'X';

    std::vector<Node*> transition = std::vector<Node*>(128, nullptr);
};

Node* root = new Node();
size_t stringIndex = 0;
std::vector<Node*> nodes;

```

```
public:
TrieAutomaton();
~TrieAutomaton();

void add_string(std::string const & s);
void construct_automaton();

std::vector<std::vector<size_t>> search(std::string const & s);


};

void TrieAutomaton::add_string(std::string const & s){
    Node* at = root;

    for(char c : s){
        if(at->transition[c] == nullptr){
            Node* newNode = new Node();
            nodes.push_back(newNode);
            newNode->depth = at->depth + 1;
            newNode->nodeChar = c;
            at->transition[c] = newNode;
        }
         at = at->transition[c];
    }

    at->terminal.push_back(stringIndex);
    stringIndex++;
}

void TrieAutomaton::construct_automaton(){
    Node* at;
    std::queue<Node*> bfsQ;
    bfsQ.push(root);

    while(!bfsQ.empty()){
        at = bfsQ.front();
        bfsQ.pop();
        for(unsigned char c = 0; c < 128; c++){
            if(at->transition[c] == nullptr){
                if(at == root) at->transition[c] = root;
```

```
                else at->transition[c] = at->suffixNode->transition[c];

            }
            else{
                if(at == root) at->transition[c]->suffixNode = root;
                else at->transition[c]->suffixNode = at->suffixNode->transition[c]; //Could probably make some chain here during construction.

    //Suffix chain
    if(!at->suffixNode->transition[c]->terminal.empty()){
        at->transition[c]->terminalChain = at->suffixNode->transition[c];
    }
    else{
        at->transition[c]->terminalChain = at->suffixNode->transition[c]->terminalChain;
    }

    if(at == root){
        at->transition[c]->terminalChain = root;
    }
                bfsQ.push(at->transition[c]);
            }
        }
    }
}

std::vector<std::vector<size_t» TrieAutomaton::search(std::string const & s){
    Node* at = root;
    std::vector<std::vector<size_t» ans(stringIndex);
    for(size_t i = 0; i < s.length(); i++){
        char c = s[i];
        at = at->transition[c];

        Node* suffixLinking = at;
        Node* prev = nullptr;
        while(suffixLinking != root){  //Somehow safe the previous terminal suffix for all nodes to save time.
            if(suffixLinking == prev) std::cerr « "Linking err";

            for(auto index : suffixLinking->terminal){
                ans[index].push_back(i - suffixLinking->depth + 1);
            }

            prev = suffixLinking;
            suffixLinking = suffixLinking->terminalChain;
        }
    }
    return ans;
}

TrieAutomaton::TrieAutomaton(){
    root->suffixNode = root;
    root->depth = 0;
    root->terminalChain = root;
}

TrieAutomaton::~TrieAutomaton(){
    delete root;
    for(auto x : nodes){
        delete x;
    }
}
#endif
```

5.56 Strings/Suffix-sorting/Suffix_Sorting.hpp File Reference

```
#include <string>
#include <algorithm>
#include <vector>
#include <iostream>
#include <stddef.h>
```

Classes

- class SuffixArray

5.57 Suffix_Sorting.hpp

Go to the documentation of this file.

```
/*
AUTHOR: Oliver Lindgren
*/

#ifndef SUFFIX_ARRAY_HPP
#define SUFFIX_ARRAY_HPP

#include<string>
#include<algorithm>
#include<vector>
#include<iostream>
#include<stddef.h>

/*
A class that computes the suffix-array of a string s.
Takes a string to compute the array for as argument to the constructor.
Also provides a function getSuffix(i) which returns the index in s at which the i:th smallest suffix
begins.

Time complexity of the constructor is O(n log^2(n)),
getSuffix runs in constant time.
*/
class SuffixArray{
private:
    std::vector<size_t> sorted;

public:
    SuffixArray(std::string const & s);
    size_t getSuffix(size_t i);

};





























SuffixArray::SuffixArray(std::string const & s){
    std::vector<size_t> buckets(s.length());
    std::vector<size_t> newBuckets(s.length());
    std::vector<size_t> suffixPosition(s.length());

    buckets[0] = 0;
    newBuckets[0] = 0;

    for(size_t i = 0; i < s.size(); i++){
        sorted.push_back(i);
    }

    //Do an initial sorting on just the first charachter
    std::sort(sorted.begin(), sorted.end(), [&s](const size_t a, const size_t b){
        return s[a] < s[b];
    });

    buckets[0] = 0;
    suffixPosition[sorted[0]] = 0;

    for(int i = 1; i < s.length(); i++){
        if(s[sorted[i]] == s[sorted[i-1]]) buckets[i] = buckets[i-1];
        else buckets[i] = buckets[i-1] + 1;
```

```
        suffixPosition[sorted[i]] = buckets[i];
    }



    //This loop will run log(n) times
    for(int k = 1; k <= s.length(); k *= 2){

        //Sorting takes n log(n) time
        std::sort(sorted.begin(), sorted.end(), [&s, &k, &suffixPosition](const size_t a, const size_t b){
            if(suffixPosition[a] == suffixPosition[b]){
                //If one of the suffixes in this buckets end then it should come before the others
                if(a+k >= s.length() || b+k >= s.length()){
                    return a > b; // '>' instead of '<' because the bigger the number, the shorter the suffix
                }
                //Otherwise we sort based on the next 2^k chars, which have already been sorted somewhere else.
                return suffixPosition[a+k] < suffixPosition[b+k];
            }
            //If they don't belong to the same bucket then their relative position shall remain the same.
            return suffixPosition[a] < suffixPosition[b];
        });

        //This will run n times for bookkeeping
        for(int i = 1; i < s.length(); i++){
            if(buckets[i] > buckets[i-1]) newBuckets[i] = newBuckets[i-1] + 1;
            else if(suffixPosition[sorted[i] + k] > suffixPosition[sorted[i-1] + k]) newBuckets[i] = newBuckets[i-1] + 1;
            else newBuckets[i] = newBuckets[i-1];
        }

        buckets.swap(newBuckets);

        for(int i = 0; i < s.length(); i++){
            suffixPosition[sorted[i]] = buckets[i];
        }

    }
}

size_t SuffixArray::getSuffix(size_t i){
    return sorted[i];
}

#endif
```

Index