

Algo_Llb

Generated by Doxygen 1.9.8

1 Algorithm_Library	1
2 Class Index	3
2.1 Class List	3
3 File Index	5
3.1 File List	5
4 Class Documentation	7
4.1 DisjointSet Class Reference	7
4.1.1 Constructor & Destructor Documentation	7
4.1.1.1 DisjointSet()	7
4.1.2 Member Function Documentation	7
4.1.2.1 query()	7
4.1.2.2 unionSets()	8
4.2 Edge Struct Reference	8
4.2.1 Member Data Documentation	8
4.2.1.1 capacity	8
4.2.1.2 destination	8
4.2.1.3 firstDeparture	8
4.2.1.4 flow	8
4.2.1.5 from	9
4.2.1.6 repeatDepartures	9
4.2.1.7 reverse	9
4.2.1.8 to	9
4.2.1.9 travelTime	9
4.2.1.10 weight	9
4.3 FenwickTree Class Reference	9
4.3.1 Constructor & Destructor Documentation	10
4.3.1.1 FenwickTree()	10
4.3.1.2 ~FenwickTree()	10
4.3.2 Member Function Documentation	10
4.3.2.1 Query()	10
4.3.2.2 Update()	10
4.4 Point< T > Struct Template Reference	10
4.4.1 Member Function Documentation	11
4.4.1.1 length()	11
4.4.1.2 lengthSquared()	11
4.4.1.3 magnitude()	11
4.4.1.4 magnitudeSquared()	11
4.4.1.5 operator"!="()	11
4.4.1.6 operator+()	11
4.4.1.7 operator-()	11

4.4.1.8 operator==()	12
4.4.2 Member Data Documentation	12
4.4.2.1 x	12
4.4.2.2 y	12
4.5 Prime_Sieve Class Reference	12
4.5.1 Constructor & Destructor Documentation	12
4.5.1.1 Prime_Sieve()	12
4.5.2 Member Function Documentation	12
4.5.2.1 get_Number_Of_Primes()	12
4.5.2.2 get_Primes()	13
4.5.2.3 is_Prime()	13
4.6 Rational< T > Struct Template Reference	13
4.6.1 Member Function Documentation	13
4.6.1.1 normalise()	13
4.6.1.2 operator*()	13
4.6.1.3 operator+()	14
4.6.1.4 operator-()	14
4.6.1.5 operator/()	14
4.6.2 Member Data Documentation	14
4.6.2.1 denominator	14
4.6.2.2 numerator	14
4.7 Segment_Tree< T > Class Template Reference	14
4.7.1 Constructor & Destructor Documentation	15
4.7.1.1 Segment_Tree()	15
4.7.2 Member Function Documentation	15
4.7.2.1 Query()	15
4.8 SuffixArray Class Reference	15
4.8.1 Constructor & Destructor Documentation	15
4.8.1.1 SuffixArray()	15
4.8.2 Member Function Documentation	15
4.8.2.1 getSuffix()	15
4.9 TrieAutomaton Class Reference	16
4.9.1 Constructor & Destructor Documentation	16
4.9.1.1 TrieAutomaton()	16
4.9.1.2 ~TrieAutomaton()	16
4.9.2 Member Function Documentation	16
4.9.2.1 add_string()	16
4.9.2.2 construct_automaton()	16
4.9.2.3 search()	16
5 File Documentation	17
5.1 Arithmetic/Chinese-Remainder-Theorem/Chinese_Remainder.hpp File Reference	17

5.1.1 Function Documentation	17
5.1.1.1 Solve_System_Of_Congruencies()	17
5.2 Chinese_Remainder.hpp	18
5.3 Arithmetic/Modular-Arithmetic/modular.hpp File Reference	18
5.3.1 Typedef Documentation	19
5.3.1.1 i128	19
5.3.1.2 ll	19
5.3.1.3 u128	19
5.3.1.4 ull	19
5.3.2 Function Documentation	19
5.3.2.1 gcd()	19
5.3.2.2 mod()	19
5.3.2.3 modAdd()	19
5.3.2.4 modDiv()	20
5.3.2.5 modInverse()	20
5.3.2.6 modMult()	20
5.3.2.7 modSub()	20
5.4 modular.hpp	20
5.5 Arithmetic/Rational arithmetic/rational.hpp File Reference	21
5.5.1 Function Documentation	22
5.5.1.1 gcd()	22
5.5.1.2 operator<<()	22
5.5.1.3 operator>>()	22
5.6 rational.hpp	22
5.7 Data_Structures/Disjoint_Set/DisjointSet.hpp File Reference	23
5.8 DisjointSet.hpp	24
5.9 Data_Structures/FenwickTree/FenwickTree.hpp File Reference	25
5.10 FenwickTree.hpp	25
5.11 Data_Structures/Segment_Tree/Segment_Tree.hpp File Reference	26
5.12 Segment_Tree.hpp	26
5.13 Geometry/Convex_Hull/Convex_Hull.hpp File Reference	27
5.13.1 Function Documentation	27
5.13.1.1 convex_hull()	27
5.14 Convex_Hull.hpp	28
5.15 Geometry/Inside_Polygon/Inside.hpp File Reference	28
5.15.1 Enumeration Type Documentation	29
5.15.1.1 Status	29
5.15.2 Function Documentation	29
5.15.2.1 inside_poly()	29
5.15.2.2 myAbs()	29
5.15.2.3 onSegment()	30
5.16 Inside.hpp	30

5.17 Geometry/Line_Intersection/intersection.hpp File Reference	31
5.17.1 Function Documentation	31
5.17.1.1 line_intersection()	31
5.18 intersection.hpp	31
5.19 Geometry/Points/Points.hpp File Reference	32
5.19.1 Function Documentation	32
5.19.1.1 add()	32
5.19.1.2 angle()	33
5.19.1.3 cross_product()	33
5.19.1.4 dist()	33
5.19.1.5 distSquared()	33
5.19.1.6 dot_product()	33
5.19.1.7 intersect()	33
5.19.1.8 length()	33
5.19.1.9 lengthSquared()	34
5.19.1.10 operator<<()	34
5.19.1.11 operator>>()	34
5.19.1.12 scalar_multiplication()	34
5.19.1.13 simple_polygon()	34
5.19.1.14 subtract()	34
5.20 Points.hpp	35
5.21 Geometry/Polygon_Area/Area.hpp File Reference	36
5.21.1 Function Documentation	37
5.21.1.1 Polygon_area()	37
5.22 Area.hpp	37
5.23 Graphs/Bellman_Ford/Bellman_Ford.hpp File Reference	37
5.23.1 Function Documentation	38
5.23.1.1 Bellman_Ford()	38
5.23.2 Variable Documentation	38
5.23.2.1 inf	38
5.23.2.2 negInf	38
5.24 Bellman_Ford.hpp	38
5.25 Graphs/Dijkstra/shortest_path.hpp File Reference	39
5.25.1 Macro Definition Documentation	40
5.25.1.1 distance	40
5.25.1.2 node	40
5.25.1.3 parent	40
5.25.1.4 weight	40
5.25.2 Typedef Documentation	40
5.25.2.1 ull	40
5.25.3 Function Documentation	40
5.25.3.1 dijkstra_shortest_path()	40

5.26 shortest_path.hpp	41
5.27 Graphs/Dijkstra_Time_Table/shortest_path_time_table.hpp File Reference	41
5.27.1 Macro Definition Documentation	42
5.27.1.1 distance	42
5.27.1.2 node	42
5.27.1.3 parent	42
5.27.1.4 weight	42
5.27.2 Typedef Documentation	43
5.27.2.1 ull	43
5.27.3 Function Documentation	43
5.27.3.1 dijkstra_shortest_path_time_table()	43
5.28 shortest_path_time_table.hpp	43
5.29 Graphs/Eulerian_Path/Eulerian_Path.hpp File Reference	44
5.29.1 Function Documentation	44
5.29.1.1 Eulerian_Path_Directed()	44
5.29.1.2 Eulerian_Path_Undirected()	44
5.30 Eulerian_Path.hpp	45
5.31 Graphs/Floyd_Warhsall/floyd_warshall.hpp File Reference	47
5.31.1 Function Documentation	47
5.31.1.1 Floyd_Warshall()	47
5.32 floyd_warshall.hpp	47
5.33 Graphs/Maximum_Flow/Maximum_Flow.hpp File Reference	48
5.33.1 Function Documentation	48
5.33.1.1 dinic_bfs()	48
5.33.1.2 dinic_dfs()	49
5.33.1.3 dinic_max_flow()	49
5.34 Maximum_Flow.hpp	49
5.35 Graphs/Minimum_Cut/Minimum_Cut.hpp File Reference	51
5.35.1 Function Documentation	51
5.35.1.1 minimum_cut()	51
5.36 Minimum_Cut.hpp	52
5.37 Graphs/MST/MST.hpp File Reference	52
5.37.1 Function Documentation	53
5.37.1.1 mst()	53
5.38 MST.hpp	53
5.39 Misc/Equation solver plus/solver.hpp File Reference	53
5.39.1 Function Documentation	54
5.39.1.1 equation_solver()	54
5.39.1.2 is_double_zero()	54
5.40 solver.hpp	54
5.41 Misc/Interval Cover/cover.hpp File Reference	56
5.41.1 Function Documentation	56

5.41.1.1 cover()	56
5.42 cover.hpp	56
5.43 Misc/Knapsack/Knapsack.hpp File Reference	57
5.43.1 Function Documentation	57
5.43.1.1 knapsack()	57
5.44 Knapsack.hpp	58
5.45 Misc/Longest Increasing Subsequence/lis.hpp File Reference	58
5.45.1 Function Documentation	59
5.45.1.1 lis()	59
5.46 lis.hpp	59
5.47 Misc/Polymul/FFT.hpp File Reference	60
5.47.1 Macro Definition Documentation	60
5.47.1.1 PI	60
5.47.2 Typedef Documentation	60
5.47.2.1 ll	60
5.47.3 Function Documentation	60
5.47.3.1 FFT()	60
5.47.3.2 INVERSE_FFT()	61
5.47.3.3 Polymul()	61
5.47.3.4 Polymul_Quadratic()	61
5.48 FFT.hpp	61
5.49 Prime_Numbers/Prime_Factorisation/Factorize.hpp File Reference	63
5.49.1 Typedef Documentation	64
5.49.1.1 ll	64
5.49.1.2 u128	64
5.49.1.3 ull	64
5.49.2 Function Documentation	64
5.49.2.1 f()	64
5.49.2.2 fast_factors()	64
5.49.2.3 fast_prime()	64
5.49.2.4 gcd()	64
5.49.2.5 is_Probably_Prime()	65
5.49.2.6 Miller_test()	65
5.49.2.7 modMult()	65
5.49.2.8 nested()	65
5.49.2.9 Pollard_Rho()	65
5.49.2.10 powMod()	65
5.49.2.11 prime_with_mult()	65
5.49.3 Variable Documentation	66
5.49.3.1 primes	66
5.50 Factorize.hpp	66
5.51 Prime_Numbers/Prime_Sieve/Sieve.hpp File Reference	68

5.52 Sieve.hpp	68
5.53 README.md File Reference	70
5.54 Strings/Aho-corasick/Aho-Corasick.hpp File Reference	70
5.55 Aho-Corasick.hpp	70
5.56 Strings/Suffix-sorting/Suffix_Sorting.hpp File Reference	72
5.57 Suffix_Sorting.hpp	73
Index	75

Chapter 1

Algorithm_Library

Chapter 2

Class Index

2.1 Class List

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

DisjointSet	7
Edge	8
FenwickTree	9
Point< T >	10
Prime_Sieve	12
Rational< T >	13
Segment_Tree< T >	14
SuffixArray	15
TrieAutomaton	16

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Arithmetic/Chinese-Remainder-Theorem/ Chinese_Remainder.hpp	17
Arithmetic/Modular-Arithmetic/ modular.hpp	18
Arithmetic/Rational arithmetic/ rational.hpp	21
Data_Structures/Disjoint_Set/ DisjointSet.hpp	23
Data_Structures/FenwickTree/ FenwickTree.hpp	25
Data_Structures/Segment_Tree/ Segment_Tree.hpp	26
Geometry/Convex_Hull/ Convex_Hull.hpp	27
Geometry/Inside_Polygon/ Inside.hpp	28
Geometry/Line_Intersection/ intersection.hpp	31
Geometry/Points/ Points.hpp	32
Geometry/Polygon_Area/ Area.hpp	36
Graphs/Bellman_Ford/ Bellman_Ford.hpp	37
Graphs/Dijkstra/ shortest_path.hpp	39
Graphs/Dijkstra_Time_Table/ shortest_path_time_table.hpp	41
Graphs/Eulerian_Path/ Eulerian_Path.hpp	44
Graphs/Floyd_Warshall/ floyd_warshall.hpp	47
Graphs/Maximum_Flow/ Maximum_Flow.hpp	48
Graphs/Minimum_Cut/ Minimum_Cut.hpp	51
Graphs/MST/ MST.hpp	52
Misc/Equation solver plus/ solver.hpp	53
Misc/Interval Cover/ cover.hpp	56
Misc/Knapsack/ Knapsack.hpp	57
Misc/Longest Increasing Subsequence/ lis.hpp	58
Misc/Polymul/ FFT.hpp	60
Prime_Numbers/Prime_Factorisation/ Factorize.hpp	63
Prime_Numbers/Prime_Sieve/ Sieve.hpp	68
Strings/Aho-corasick/ Aho-Corasick.hpp	70
Strings/Suffix-sorting/ Suffix_Sorting.hpp	72

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

$x = a_1 \bmod b_1$ $x = a_2 \bmod b_2$ $x = a_n \bmod b_n$

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

b_1 - b_n do not have to be co-prime //ACTUALLY THEY DO, BUT THIS WILL BE FIXED IN THE FUTURE If you know b_1 - b_n 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.](#)

```

00001 #ifndef CHINESE_REMAINDER_HPP
00002 #define CHINESE_REMAINDER_HPP
00003 #include<vector>
00004 #include<utility>
00005 #include"Arithmetic\Modular-Arithmetic\modular.hpp"
00006
00026 long long int Solve_System_Of_Congruencies(std::vector<std::pair<long long int, long long int>
    equations, bool co_prime = false){
00027
00028     if(co_prime) goto start_solving;
00029
00030
00031
00032     start_solving:
00033
00034     long long int lcm = 1;
00035     for(auto& pair : equations){
00036         lcm *= pair.second;
00037     }
00038
00039     long long int ans = 0;
00040
00041     for(auto& pair : equations){
00042         long long int Mi = lcm/pair.second;
00043         long long int xi = Mi % pair.second;
00044         long long int bi = modInverse(xi, pair.second);
00045         long long int yi = (bi * Mi) % lcm;
00046         ans = modAdd(ans, modMult(pair.first, yi, lcm), lcm);
00047     }
00048
00049     return ans;
00050 }
00051
00052 #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.](#)

```
00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #include<utility>
00006 #include<cstdlib>
00007 #include<stddef.h>
00008
00009
00010 typedef long long int ll;
00011 typedef unsigned long long int ull;
00012 typedef __uint128_t ul128;
00013 typedef __int128_t i128;
00014
00015 //Returnerar a mod m. Fungerar för både negativa och positiva tal a.
00016 inline ll mod(const i128 a, const i128 m){
00017     return (m+(a%m))%m;
00018 }
00019
00020 //Returnerar (a*b) mod m. Där a,b,m får plats i ett 64-bitars tal
00021 inline ll modMult(const i128 a, const i128 b, const i128 m){
00022     return mod(a * b, m);
00023 }
00024
00025 //Returnerar (a+b) mod m. Där a,b,m får plats i ett 64-bitars tal
00026 inline ll modAdd(const i128 a, const i128 b, const i128 m){
00027     return mod(a + b, m);
00028 }
00029
00030 //Returnerar (a-b) mod m. Där a,b,m får plats i ett 64-bitars tal
```

```

00031 inline ll modSub(const i128 a, const i128 b, const i128 m) {
00032     return mod(a - b, m);
00033 }
00034
00035 //Returnerar största gemensamma nämnaren till a och b
00036 i128 gcd(i128 a, i128 b, i128& x0, i128& y0) {
00037     a = std::abs(a);
00038     b = std::abs(b);
00039     if(a > b) {
00040         std::swap(a,b);
00041     }
00042
00043     if (a == 0) {
00044         x0 = 0;
00045         y0 = 1;
00046         return b;
00047     }
00048     i128 x1, y1, d;
00049     d = gcd(b % a, a, x1, y1);
00050
00051     x0 = y1 - (b/a) * x1;
00052     y0 = x1;
00053
00054     return d;
00055 }
00056
00057 //Returnerar inversen till b i Z_m
00058 i128 modInverse(const i128 b, const i128 m) {
00059     i128 x, y, d;
00060     d = gcd(b, m, x, y);
00061
00062     if(d != 1) return -1;
00063
00064     return mod(x, m);
00065 }
00066
00067 //Returnerar a * b^(-1) i Z_m
00068 ll modDiv(const i128 a, const i128 b, const i128 m) {
00069     i128 inverse = modInverse(b, m);
00070     if(inverse != -1) {
00071         return modMult(a, inverse, m);
00072     }
00073     else {
00074         return -1;
00075     }
00076 }

```

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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005 #include<iostream>
00006 #include<utility>
00007
00008 //A datatype that stores and performs operations on fractions. T must implement the +,-,*,/,«,»
00009 //In addition to that, euclids algorithm needs to work for T. It must also be possible to use T with
00010 //the std::abs function.
00011 template<typename T>
00012 struct Rational{
00013     T numerator, denominator;
00014
00015     Rational<T> operator+(const Rational<T>& other) const;
00016     Rational<T> operator-(const Rational<T>& other) const;
00017     Rational<T> operator*(const Rational<T>& other) const;
00018     Rational<T> operator/(const Rational<T>& other) const;
00019
00020     void normalise();
00021 };
00022
00023 template<typename T>
00024 T gcd(T a, T b);
00025
00026 template<typename T>
00027 std::istream & operator>>(std::istream & is, Rational<T> & m);
00028
00029 template<typename T>
00030 std::ostream & operator<<(std::ostream & os, const Rational<T> & m);
00031
00032 template<typename T>
00033 Rational<T> Rational<T>::operator+(const Rational<T>& other) const{
00034     Rational<T> sum;
00035     sum.denominator = denominator * other.denominator;
00036     sum.numerator = numerator * other.denominator + other.numerator * denominator;
```



```

00037     sum.normalise();
00038     return sum;
00039 }
00040
00041 template<typename T>
00042 Rational<T> Rational<T>::operator-(const Rational<T>& other) const{
00043     Rational<T> difference;
00044     difference.denominator = denominator * other.denominator;
00045     difference.numerator = numerator * other.denominator - other.numerator * denominator;
00046
00047     difference.normalise();
00048     return difference;
00049 }
00050
00051 template<typename T>
00052 Rational<T> Rational<T>::operator*(const Rational<T>& other) const{
00053     Rational<T> product;
00054     product.denominator = denominator * other.denominator;
00055     product.numerator = numerator * other.numerator;
00056
00057     product.normalise();
00058     return product;
00059 }
00060
00061 template<typename T>
00062 Rational<T> Rational<T>::operator/(const Rational<T>& other) const{
00063     Rational<T> swapped;
00064     swapped.denominator = other.numerator;
00065     swapped.numerator = other.denominator;
00066
00067     Rational<T> ans = *this * swapped;
00068     ans.normalise();
00069     return ans;
00070 }
00071
00072 template<typename T>
00073 void Rational<T>::normalise(){
00074     T divisor = gcd(denominator, numerator);
00075     denominator /= divisor;
00076     numerator /= divisor;
00077
00078     if(denominator < 0 && numerator < 0){
00079         denominator = -denominator;
00080         numerator = -numerator;
00081     }
00082     else if(denominator < 0){
00083         denominator = -denominator;
00084         numerator = -numerator;
00085     }
00086 }
00087
00088 template<typename T>
00089 std::istream & operator>>(std::istream & is, Rational<T>& m){
00090     is >> m.numerator >> m.denominator;
00091     m.normalise();
00092     return is;
00093 }
00094
00095 template<typename T>
00096 std::ostream & operator<<(std::ostream & os, const Rational<T> & m){
00097     os << m.numerator << " / " << m.denominator;
00098     return os;
00099 }
00100
00101 template<typename T>
00102 T gcd(T a, T b){
00103     a = std::abs(a);
00104     b = std::abs(b);
00105     if(a > b){
00106         std::swap(a,b);
00107     }
00108
00109     if (a == 0)
00110         return b;
00111     return gcd(b % a, a);
00112 }
00113

```

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.](#)

```

00001  /*
00002     AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef DISJOINTSET_HPP
00006  #define DISJOINTSET_HPP
00007
00008  #include<vector>
00009  #include<functional>
00010
00011  /*
00012   A class designed to keep track of N different sets of N total elements.
00013   Has a function to merge two sets, and a function to check if two elements are in the same set.
00014   TODO: Change to size_t
00015  */
00016  class DisjointSet {
00017  private:
00018      std::vector<int> comp;
00019
00020
00021      int representative(unsigned int a);
00022
00023  public:
00024      DisjointSet(unsigned int elements);
00025
00026      void unionSets(unsigned int a, unsigned int b);
00027
00028      bool query(unsigned int a, unsigned int b);
00029  };
00030
00031
00032
00033
00034
00035
00036
00037
00038
00039
00040
00041
00042
00043
00044
00045
00046
00047
00048
00049
00050  DisjointSet::DisjointSet(unsigned int elements) : comp(elements, -1) {};
00051
00052  void DisjointSet::unionSets(unsigned int a, unsigned int b){
00053      a = representative(a);
00054      b = representative(b);
00055
00056      if(a == b) return;
00057
00058      if(comp[a] > comp[b]) std::swap(a,b);
00059
00060      comp[a] += comp[b]; //The size of set containing a has been increased by size of set containing b.
00061      comp[b] = a; //All elements that had b as a representative will now have a as a representative.
00062  }
00063
00064  int DisjointSet::representative(unsigned int a){
00065      if(comp[a] < 0) return a;
00066
00067      //recursively find and update our representative
00068      comp[a] = representative(comp[a]);
00069      return comp[a];
00070  }
00071
00072  bool DisjointSet::query(unsigned int a, unsigned int b){
00073      return representative(a) == representative(b);

```

```

00074 }
00075
00076
00077
00078 #endif

```

5.9 Data_Structures/FenwickTree/FenwickTree.hpp File Reference

```
#include <cstdint>
```

Classes

- class [FenwickTree](#)

5.10 FenwickTree.hpp

[Go to the documentation of this file.](#)

```

00001 /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005 #ifndef FENWICKTREE_HPP
00006 #define FENWICKTREE_HPP
00007
00008 #include<cstdint>
00009
00010 //A class designed to compute and maintain the prefix sum of an array.
00011 class FenwickTree{
00012 private:
00013     long long int* base;
00014     size_t size;
00015 public:
00016     //initSize is the size of the array to be stored.
00017     FenwickTree(size_t initSize);
00018     ~FenwickTree();
00019
00020     //This increases the value of array[index] with value.
00021     void Update(size_t index, long long int value);
00022
00023     //This queries the array to get the sum of all the values in the array in the range [0, index)
00024     long long int Query(size_t index);
00025 };
00026
00027
00028
00029
00030
00031
00032
00033
00034
00035
00036
00037
00038
00039
00040
00041
00042
00043
00044
00045
00046
00047
00048
00049
00050
00051
00052
00053

```

```

00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064 FenwickTree::FenwickTree(size_t initSize){
00065     size = initSize+1;
00066     base = new long long int[size]();
00067 }
00068
00069 FenwickTree::~FenwickTree(){
00070     delete[] base;
00071 }
00072
00073
00074 void FenwickTree::Update(size_t index, long long int value){
00075     index++;
00076
00077     //Use bit indexing to update necessary values.
00078     while(index < size){
00079         base[index] += value;
00080         index += index & (-index);
00081     }
00082 }
00083
00084 long long int FenwickTree::Query(size_t index){
00085     long long int sum = 0;
00086
00087     //Use bit indexing to retrieve necessary values.
00088     while(index > 0){
00089         sum += base[index];
00090         index -= index & (-index);
00091     }
00092     return sum;
00093 }
00094 #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.](#)

```

00001 #ifndef SEGMENT_TREE_HPP
00002 #define SEGMENT_TREE_HPP
00003
00004 #include<vector>
00005 #include <functional>
00006
00007 template<class T>
00008 class Segment_Tree{
00009
00010 int leftMin, rightMax;
00011 std::vector<T> internal;
00012 std::function<T(T,T)> merge;
00013
00014 void compute(int at, int l, int r, const std::vector<T>& v){
00015     if(l == r){

```

```

00016         internal[at] = v[l];
00017         return;
00018     }
00019     int mid = (l+r)/2;
00020     compute(at*2 + 1, l, mid, v);
00021     compute(at*2 + 2, mid+1, r, v);
00022
00023     internal[at] = merge(internal[at*2+1], internal[at*2+2]);
00024 }
00025
00026 T search(int at, int l, int r, int currL, int currR){
00027     if(currL >= l && currR <= r) return internal[at];
00028
00029     int mid = (currL + currR) / 2;
00030     if(r <= mid) return search(at*2+1, l, r, currL, mid);
00031     else if(l >= mid+1) return search(at*2+2, l, r, mid+1, currR);
00032
00033     return merge(search(at*2+1, l, r, currL, mid), search(at*2+2, l, r, mid+1, currR));
00034 }
00035
00036 public:
00037
00038 Segment_Tree(const std::vector<T>& v, std::function<T(T,T)> mergeFunc){
00039     internal.resize(v.size()*4);
00040     merge = mergeFunc;
00041     leftMin = 0;
00042     rightMax = v.size()-1;
00043     compute(0, leftMin, rightMax, v);
00044 }
00045
00046 T Query(int left, int right){
00047     return search(0, left, right, leftMin, rightMax);
00048 }
00049
00050 };
00051
00052 #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.](#)

```

00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef CONVEX_HULL_HPP
00006 #define CONVEX_HULL_HPP
00007 #include "Geometry\Points\Points.hpp"
00008 #include <vector>
00009 #include <algorithm>
00010
00011 /*
00012 Function that takes a vector of points and returns the convex hull encircling them.
00013 The hull is given as a list of points in counterclockwise order, each point being a node in the hull.
00014 If the points are all colinear then the two endpoints are given.
00015
00016 Time Complexity: O(N log(n))
00017 */
00018
00019 template<typename T>
00020 std::vector<Point<T>> convex_hull(std::vector<Point<T>> points){
00021     int leftMostPoint = 0;
00022     Point<T> org = {0,0};
00023     std::vector<Point<T>> hull;
00024
00025     //This is to later remove duplicates
00026     std::sort(points.begin(), points.end(), [&org](Point<T> const & a, Point<T> const & b){
00027         if(a.x == b.x)
00028             return a.y < b.y;
00029         else
00030             return a.x < b.x;
00031     });
00032
00033     //remove duplicates
00034     std::vector<Point<T>> newPoints;
00035     newPoints.push_back(points[0]);
00036     for(int i = 1; i < points.size(); i++){
00037         if(points[i] != *(newPoints.rbegin())) newPoints.push_back(points[i]);
00038     }
00039
00040
00041     points.swap(newPoints);
00042
00043     //Find the leftmost point that is definitely on the hull
00044     for(int i = 1; i < points.size(); i++){
00045         if(points[i].x < points[leftMostPoint].x) leftMostPoint = i;
00046         else if(points[i].x == points[leftMostPoint].x && points[i].y > points[leftMostPoint].y)
00047             leftMostPoint = i;
00048     }
00049
00050     org = points[leftMostPoint];
00051     hull.push_back(points[leftMostPoint]);
00052     points.erase(points.begin() + leftMostPoint);
00053
00054     //Sort the points based on angle from the previously determined point
00055     std::sort(points.begin(), points.end(), [&org](Point<T> const & a, Point<T> const & b){
00056         T cross = cross_product(subtract(a, org), subtract(b, org));
00057         if(cross == 0) return distSquared(org, a) < distSquared(org, b); //De är ko-linjära
00058         else return cross > 0;
00059     });
00060
00061     //add points to the hull
00062     for(Point<T> & p : points){
00063         //if we are doing a rightturn pop last point of hull.
00064         while(hull.size() >= 2 && cross_product(subtract(hull[hull.size()-1], hull[hull.size()-2]),
00065             subtract(p, hull[hull.size()-2])) <= 0) hull.pop_back();
00066         hull.push_back(p);
00067     }
00068     return hull;
00069 }
00070 #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.](#)

```
00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef POLYGON_POINT_INSIDE_HPP
00006 #define POLYGON_POINT_INSIDE_HPP
00007 #include "Geometry\Points\Points.hpp"
00008 #include <vector>
00009 #include <stdexcept>
00010 #include <utility>
00011 #include <stddef.h>
00012 #include <iostream>
00013 #include <cmath>
00014
00015 enum Status{Outside, OnEdge, Inside};
00016
00017 //For some reason the normal abs function didn't work
00018 double myAbs(double y){
00019     if(y < 0) return -y;
00020     else return y;
00021 }
00022
00023
00024 /*
00025 Function to determine if a point lies on a segment.
00026 Takes as input a pair containing the two endPoints of the segment, as well as a point.
00027 Returns true if the point lies on the segment.
00028
00029 Should probably take some sort of epsilon as a value too
00030 */
00031 template <typename T>
00032 bool onSegment(std::pair<Point<T>, Point<T>> segment, Point<T> q){
00033     auto x = dist(segment.first, segment.second);
00034     auto a = dist(q, segment.first);
00035     auto b = dist(q, segment.second);
00036     auto y = x - a - b;
00037     double t = myAbs(y); //For some reason abs(y) returned 0 even when y == 0.7.... something
00038     return t < 0.00001;
00039 }
00040
00041 /*
00042 Function to determine if a point is inside, outside, or on the edge of a simple polygon.
00043 Takes as input a vector containing the points of the simple polygon and a point p.
00044 Returns a Status enum.
00045 If the point is inside the polygon Status::Inside is returned.
00046 If the point is outside the polygon Status::Outside is returned.
00047 If the point is on the edge of the polygon Status::OnEdge is returned.
00048 Time complexity: O(N)
00049 */
00050 template <typename T>
00051 Status inside_poly(std::vector<Point<T>> const & polygon, Point<T> const & p){
00052     double angleSum = 0;
00053     size_t n = polygon.size();
00054     for(size_t i = 0; i < n; i++){
00055         if(onSegment({polygon[i], polygon[(i+1)%n]}, p)){
00056             std::cerr << i << ' ' << ((i+1)%n) << '\n';
00057             return OnEdge;
00058         }
00059         angleSum += angle(subtract(polygon[i], p), subtract(polygon[(i+1)%n], p));
00060     }
00061
00062     if(abs(angleSum) < 0.00001) return Outside;
00063     if(abs(abs(angleSum) - 2.0*(3.141592)) < 00001) return Inside;
00064
00065     throw std::logic_error("Can't determine point position");
00066 }
00067 }
00068
00069 #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.](#)

```
00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef LINE_INTERSECTION_HPP
00006 #define LINE_INTERSECTION_HPP
00007
00008 #include "Geometry\Points\Points.hpp"
00009 #include <vector>
00010 #include <utility>
00011
00012 template<typename T>
00013 std::vector<std::pair<Point<double>, Point<double>>> line_intersection(Point<T> p1, Point<T> p2,
00014 Point<T> q1, Point<T> q2){
00015     Point<T> deltaP = subtract(p2, p1);
00016     Point<T> deltaQ = subtract(q2, q1);
00017
00018     T cross = cross_product(deltaP, deltaQ);
00019     if(cross == 0){
00020         Point<T> x = subtract(q1, p1);
00021         if(cross_product(x, deltaP) == 0) return{p1, p2};
00022         else return {};
00023     }
00024     else{
00025         double s = (double) cross_product(subtract(q1, p1), deltaP) / (double)cross;
00026         double t = (double) cross_product(subtract(p1, q1), deltaQ) / (double)cross;
00027     }
00028 }
00029
00030 #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.](#)

```

00001  /*
00002  Author: Oliver Lindgren
00003  */
00004
00005  #ifndef POINTS_HPP
00006  #define POINTS_HPP
00007  #include <cmath>
00008  #include <iostream>
00009  #include <vector>
00010
00011  template<typename T>
00012  struct Point{
00013      T x, y;
00014
00015      double length(){
00016          return sqrt(x*x + y*y);
00017      }
00018
00019      double magnitude(){
00020          return length();
00021      }
00022
00023      T lengthSquared(){
00024          return x*x + y*y;
00025      }
00026
00027      T magnitudeSquared(){
00028          return lengthSquared();
00029      }
00030
00031      bool operator==(Point<T> const & other){
00032          return x == other.x && y == other.y;
00033      }
00034
00035      bool operator!=(Point<T> const & other){
00036          return x != other.x || y != other.y;
00037      }
00038
00039      Point<T> operator+(Point<T> const & other){
00040          return {x + other.x, y + other.y};
00041      }
00042
00043      Point<T> operator-(Point<T> const & other){
00044          return {x - other.x, y - other.y};
00045      }
00046  };
00047
00048  template<typename T>
00049  std::ostream& operator<< (std::ostream& out, Point<T>& point){
00050      out << point.x << ' ' << point.y;
00051      return out;
00052  }
00053
00054  template<typename T>
00055  std::istream& operator>> (std::istream& in, Point<T>& point){
00056      in >> point.x >> point.y;
00057      return in;
00058  }
00059
00060  template<typename T>
00061  Point<T> add(Point<T> const & a, Point<T> const & b){
00062      return {a.x + b.x, a.y + b.y};
00063  }
00064
00065  template<typename T>
00066  Point<T> subtract(Point<T> const & a, Point<T> const & b){
00067      return {a.x - b.x, a.y - b.y};
00068  }
00069
00070  template<typename T>
00071  Point<T> scalar_multiplication(Point<T> const & point, T scalar){
00072      return {point.x * scalar, point.y * scalar};
00073  }
00074
00075  template<typename T>
00076  T dot_product(Point<T> const & a, Point<T> const & b){
00077      return a.x * b.x + a.y * b.y;
00078  }
00079
00080  template<typename T>
00081  T cross_product(Point<T> const & a, Point<T> const & b){
00082      return a.x * b.y - b.x * a.y;

```

```

00083 }
00084
00085 template<typename T>
00086 double length(Point<T> const & a){
00087     return sqrt(a.x*a.x + a.y*a.y);
00088 }
00089
00090 template<typename T>
00091 double lengthSquared(Point<T> const & a){
00092     return a.x*a.x + a.y*a.y;
00093 }
00094
00095 template<typename T>
00096 double dist(Point<T> const & a, Point<T> const & b){
00097     return length(subtract(a, b));
00098 }
00099
00100 template<typename T>
00101 double distSquared(Point<T> const & a, Point<T> const & b){
00102     return lengthSquared(subtract(a, b));
00103 }
00104
00105 template<typename T>
00106 double angle(Point<T> const & a, Point<T> const & b){
00107     return atan2(cross_product(a,b), dot_product(a,b));
00108 }
00109
00110 template<typename T>
00111 bool intersect(Point<T> const & p1, Point<T> const & p2, Point<T> const & q1, Point<T> const & q2){
00112     return (((p1.x-q1.x) * (p2.y - q1.y)) - ((p2.x-q1.x) * (p1.y - q1.y)) > 0) == //First
determinant
00113         (((p1.x-q2.x) * (p2.y - q2.y)) - ((p2.x-q2.x) * (p1.y - q2.y)) < 0) && //Second
00114         (((q1.x-p1.x) * (q2.y - p1.y)) - ((q2.x-p1.x) * (q1.y - p1.y)) > 0) == //Third
determinant
00115         (((q1.x-p2.x) * (q2.y - p2.y)) - ((q2.x-p2.x) * (q1.y - p2.y)) < 0)
00116     );
00117 }
00118 /* Source that this works
00119 https://stackoverflow.com/a/3842157
00120 http://www.cs.cmu.edu/~quake/robust.html
00121 */
00122
00123 template<typename T>
00124 bool simple_polygon(std::vector<Point<T>> const & v){
00125     size_t n = v.size();
00126     bool ans = true;
00127     for(int i = 2; i < n; i++){
00128         for(int j = (i == 6 ? 1 : 0); j < i-1; j++){
00129             ans &= !intersect(v[i], v[(i+1)%n], v[j], v[(j+1)%n]);
00130         }
00131     }
00132     return ans;
00133 }
00134
00135 #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.](#)

```
00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef POLYGON_AREA_HPP
00006 #define POLYGON_AREA_HPP
00007 #include "Geometry\Points\Points.hpp"
00008 #include <vector>
00009 #include <stddef.h>
00010
00011 /*
00012 Computes the area of a simple polygon.
00013 Takes as input a vector of points, the points of the polygon.
00014 Returns the area of that polygon
00015 Area is positive if points are given in counter-clockwise order.
00016 Area is negative if points are given in clockwise order.
00017
00018 Time complexity: O(N)
00019 */
00020 template<typename T>
00021 double Polygon_area(std::vector<Point<T>> points){
00022     T area = 0;
00023     size_t n = points.size();
00024     for(size_t i = 0; i < n; i++){
00025         area += cross_product(points[i], points[(i+1)%n]);
00026     }
00027     return double(area) / 2.0;
00028 }
00029
00030 #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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005 #ifndef BELLMAN_FORD_HPP
00006 #define BELLMAN_FORD_HPP
00007 #include<vector>
00008 #include<utility>
00009 #include<limits>
00010 #include<stddef.h>
00011
00012 struct Edge{
00013     size_t from, to;
00014     long long int weight;
00015 };
00016
00017 const long long int inf = std::numeric_limits<long long int>::max();
00018 const long long int negInf = std::numeric_limits<long long int>::min();
00019
00020
00021 /*
00022 My implementation of the Bellman-Ford algorithm. Calculates the distance from startNode to all other
00023 nodes in the graph.
00024 Returns a vector of pairs where the first item of each element i is the distance to the node i and the
00025 second item is the node you go to i from.
00026 If there is no way to reach a node i then v[i].first is set to inf.
00027 If node i is part of a negative cycle then v[i].first is set to negInf.
00028 The start node's parent is set to inf unless it is part of a negative cycle.
00029 Takes as input the amount of nodes in the graph, which node to compute the distance from, and a vector
00030 containing all Edges in the graph.
00031 Time complexity: O(n*m) where n is the amount of nodes in the graph and m is the amount of edges.
00032  */
```



```

00030 std::vector<std::pair<long long int, size_t> Bellman_Ford(const size_t nodes, const size_t startNode,
    const std::vector<Edge>& edges){
00031     std::vector<std::pair<long long int, size_t> dist_pred(nodes, {inf, inf});
00032
00033     dist_pred[startNode].first = 0;
00034
00035     //Construct solution
00036     for(size_t i = 0; i < nodes-1; i++){
00037         bool change = false;
00038         for(Edge e : edges){
00039             if(dist_pred[e.from].first == inf) continue; //Avoid overflow error
00040             if(dist_pred[e.from].first + e.weight < dist_pred[e.to].first){
00041                 dist_pred[e.to].first = dist_pred[e.from].first + e.weight;
00042                 dist_pred[e.to].second = e.from;
00043                 change = true;
00044             }
00045         }
00046         if(!change) return dist_pred;
00047     }
00048
00049
00050     //Now do the same thing again. Every single node that still has a shorter path to it is part of a
    negative cycle.
00051     for(size_t i = 0; i < nodes-1; i++){
00052         bool change = false;
00053         for(Edge e : edges){
00054             if(dist_pred[e.from].first == inf) continue; //Avoid overflow error
00055             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)){
00056                 dist_pred[e.to].first = negInf;
00057                 dist_pred[e.to].second = e.from;
00058                 change = true;
00059             }
00060         }
00061         if(!change) return dist_pred;
00062     }
00063
00064     return dist_pred;
00065 }
00066
00067
00068 #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 > > > &adjacency_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 > > > & adjacency_list,
    size_t startNode )
```

5.26 shortest_path.hpp

[Go to the documentation of this file.](#)

```

00001  /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef SHORTEST_PATH_DIJKSTRA_HPP
00006  #define SHORTEST_PATH_DIJKSTRA_HPP
00007  #include<vector>
00008  #include<utility>
00009  #include<queue>
00010  #include<limits>
00011  #include<stddef.h>
00012
00013
00014  /*
00015  My implementation of the dijkstra algorithm.
00016  Returns a vector of pairs where the second item of each element i is the distance to the node i and
00017  the first item is the node you go to i from.
00018  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
00019  and unsigned long long int respectively.
00020  The start node's parent is set to the maximum value of size_t
00021  Takes as input the graph represented in adjacency list form and the node we start from.
00022  Time complexity: O(m * log(n)) where n is the amount of nodes in the graph and m is the amount of
00023  edges.
00024  */
00025
00026  typedef unsigned long long int ull;
00027  #define node first
00028  #define parent first
00029  #define distance second
00030  #define weight second
00031
00032  std::vector<std::pair<size_t, ull> dijkstra_shortest_path(const
00033  std::vector<std::vector<std::pair<size_t, ull>>& adjacency_list, size_t startNode){
00034
00035      struct Compare{
00036          bool operator() (std::pair<size_t, ull> a, std::pair<size_t, ull> b){
00037              return a.distance > b.distance;
00038          }
00039      };
00040
00041      std::vector<std::pair<size_t, ull> parent_dist_pair(adjacency_list.size(),
00042      {std::numeric_limits<size_t>::max(), std::numeric_limits<ull>::max()});
00043      std::priority_queue<std::pair<size_t, ull>, std::vector<std::pair<size_t, ull>, Compare> pq;
00044
00045      pq.push({startNode, 0});
00046      parent_dist_pair[startNode].distance = 0;
00047
00048      while(!pq.empty()){
00049          std::pair<size_t, ull> p = pq.top();
00050          pq.pop();
00051
00052          if(p.distance != parent_dist_pair[p.node].distance)
00053              continue;
00054
00055          for(auto edge : adjacency_list[p.node]){
00056              if(parent_dist_pair[edge.node].distance > p.distance + edge.weight) {
00057                  pq.push({edge.node, p.distance + edge.weight});
00058                  parent_dist_pair[edge.node].distance = p.distance + edge.weight;
00059                  parent_dist_pair[edge.node].parent = p.node;
00060              }
00061          }
00062      }
00063
00064      return parent_dist_pair;
00065  }
00066
00067  #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](#) > > &adjacency_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 > > & adjacency_list,
    size_t startNode )
```

5.28 shortest_path_time_table.hpp

[Go to the documentation of this file.](#)

```
00001 /*
00002 AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef SHORTEST_PATH_TIME_TABLE_DIJKSTRA_HPP
00006 #define SHORTEST_PATH_TIME_TABLE_DIJKSTRA_HPP
00007 #include<vector>
00008 #include<utility>
00009 #include<queue>
00010 #include<limits>
00011 #include<iostream>
00012 #include<stddef.h>
00013
00014
00015 struct Edge
00016 {
00017     size_t destination;
00018     unsigned long long int travelTime;
00019     unsigned long long int firstDeparture;
00020     unsigned long long int repeatDepartures;
00021 };
00022
00023
00024 /*
00025 My implementation of the dijkstra algorithm but were certain edges may only be used on certain times.
00026 Returns a vector of pairs where the second item of each element i is the distance to the node i and
00027 the first item is the node you go to i from.
00028 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
00029 and unsigned long long int respectively.
00030 The start node's parent is set to the maximum value of size_t
00031 Takes as input the graph represented in adjacency list form and the node we start from.
00032 Time complexity: O(m * log(n)) where n is the amount of nodes in the graph and m is the amount of
00033 edges.
00034 */
00035
00036 typedef unsigned long long int ull;
00037 #define node first
00038 #define parent first
00039 #define distance second
00040 #define weight second
00041
00042 std::vector<std::pair<size_t, unsigned long long int>> dijkstra_shortest_path_time_table(const
std::vector<std::vector<Edge>& adjacency_list, size_t startNode){
00043     struct Compare{
00044         bool operator() (std::pair<size_t, ull> a, std::pair<size_t, ull> b){
00045             return a.distance > b.distance;
00046         }
00047     };
00048
00049     std::vector<std::pair<size_t, ull>> parent_dist_pair(adjacency_list.size(),
{std::numeric_limits<size_t>::max(), std::numeric_limits<ull>::max()});
```

```

00047     std::priority_queue<std::pair<size_t, ull>, std::vector<std::pair<size_t, ull>, Compare> pq;
00048
00049     pq.push({startNode, 0});
00050     parent_dist_pair[startNode].distance = 0;
00051
00052     while(!pq.empty()){
00053         std::pair<size_t, ull> p = pq.top();
00054         pq.pop();
00055
00056         if(p.distance != parent_dist_pair[p.node].distance)
00057             continue;
00058
00059         for(auto edge : adjaceny_list[p.node]){
00060             if(edge.repeatDepartures == 0 && p.distance > edge.firstDeparture) continue;
00061             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);
00062
00063             if(parent_dist_pair[edge.destination].distance > totalTime) {
00064                 pq.push({edge.destination, totalTime});
00065                 parent_dist_pair[edge.destination].distance = totalTime;
00066                 parent_dist_pair[edge.destination].parent = p.node;
00067             }
00068         }
00069     }
00070
00071     return parent_dist_pair;
00072 }
00073
00074 #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.](#)

```

00001  /*
00002      AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef EULERIAN_PATH
00006  #define EULERIAN_PATH
00007
00008  #include<vector>
00009  #include<set>
00010  #include<unordered_set>
00011  #include<stddef.h>
00012  #include<unordered_map>
00013  #include<algorithm>
00014  #include<stack>
00015  #include<iostream>
00016
00017  /*
00018  Calculates the Eulerian path in an undirected graph. Takes as input the number of nodes, as well as the
00019  edges of the graph in adjacency list form.
00020  Returns a vector of nodes, indicating the order to visit the nodes in.
00021  If no Eulerian path exists then the empty vector is returned.
00022  Time complexity: O(m) where m is the number of edges in the graph
00023  */
00024  std::vector<size_t> Eulerian_Path_Undirected(size_t nodes, const std::vector<std::vector<size_t>&
00025  neighbours);
00026
00027  /*
00028  Calculates the Eulerian path in a directed graph. Takes as input the number of nodes, as well as the
00029  edges of the graph in adjacency list form.
00030  Returns a vector of nodes, indicating the order to visit the nodes in.
00031  If no Eulerian path exists then the empty vector is returned.
00032  Time complexity: O(m) where m is the number of edges in the graph
00033  */
00034  std::vector<size_t> Eulerian_Path_Directed(size_t nodes, const std::vector<std::vector<size_t>&
00035  neighbours);
00036
00037
00038
00039
00040
00041
00042
00043
00044
00045
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055  std::vector<size_t> Eulerian_Path_Undirected(size_t nodes, const std::vector<std::vector<size_t>&
00056  neighbours){
00057      std::vector<std::unordered_map<size_t, unsigned long long int>> adjacent(nodes);
00058
00059      std::vector<size_t> path;
00060
00061      size_t edges = 0;
00062      size_t startNode = 0;
00063      size_t uneven = 0;
00064
00065      std::vector<size_t> degrees(nodes);
00066
00067      for(size_t i = 0; i < nodes; i++){
00068          for(size_t j : neighbours[i]){
00069              startNode = i;
00070              edges++;
00071              adjacent[i][j]++;
00072              degrees[i]++;
00073              degrees[j]++;
00074          }
00075      }
00076      edges /= 2;
00077      for(size_t i = 0; i < nodes; i++){

```

```

00078         degrees[i] /= 2;
00079
00080         if(degrees[i] % 2 != 0){
00081             startNode = i;
00082             uneven++;
00083         }
00084     }
00085
00086     if(uneven != 0 && uneven != 2) return {};
00087
00088     std::stack<size_t> myStack;
00089
00090     myStack.push(startNode);
00091
00092     while(!myStack.empty()){
00093         size_t v = myStack.top();
00094
00095         if(degrees[v] == 0){
00096             path.push_back(v);
00097             myStack.pop();
00098         }
00099         else{
00100             size_t next = adjacent[v].begin()->first;
00101             adjacent[v][next]--;
00102             adjacent[next][v]--;
00103             degrees[v]--;
00104             degrees[next]--;
00105             myStack.push(next);
00106             if(adjacent[v][next] == 0){
00107                 adjacent[v].erase(next);
00108                 adjacent[next].erase(v);
00109             }
00110         }
00111     }
00112     if(path.size() != edges + 1) return {};
00113     return path;
00114 }
00115
00116
00117 std::vector<size_t> Eulerian_Path_Directed(size_t nodes, const std::vector<std::vector<size_t>&
neighbours){
00118     std::vector<std::unordered_map<size_t, unsigned long long int> adjacent(nodes);
00119     std::vector<size_t> path;
00120
00121     size_t startNode = 0;
00122     size_t uneven = 0;
00123     size_t edges = 0;
00124
00125     std::vector<long long int> degrees(nodes);
00126
00127     for(size_t i = 0; i < nodes; i++){
00128         for(size_t j : neighbours[i]){
00129             startNode = i;
00130             edges++;
00131             adjacent[i][j]++;
00132             degrees[i]++;
00133             degrees[j]--;
00134         }
00135     }
00136
00137     for(size_t i = 0; i < nodes; i++){
00138         if(degrees[i] != 0){
00139             if(degrees[i] == 1)
00140                 startNode = i;
00141             uneven++;
00142         }
00143     }
00144
00145     if(uneven != 0 && uneven != 2) return {};
00146
00147     std::stack<size_t> myStack;
00148
00149     myStack.push(startNode);
00150
00151     while(!myStack.empty()){
00152         size_t v = myStack.top();
00153
00154         if(adjacent[v].empty()){
00155             path.push_back(v);
00156             myStack.pop();
00157         }
00158         else{
00159             size_t next = adjacent[v].begin()->first;
00160             adjacent[v][next]--;
00161
00162             myStack.push(next);
00163             if(adjacent[v][next] == 0){

```



```

00164         adjacent[v].erase(next);
00165     }
00166 }
00167 }
00168
00169     if(path.size() != edges+1) return {};
00170     std::reverse(path.begin(), path.end());
00171     return path;
00172 }
00173
00174 #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.](#)

```

00001 /*
00002  AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef FLOYD_WARSHALL_HPP
00006 #define FLOYD_WARSHALL_HPP
00007 #include<vector>
00008 #include<algorithm>
00009 #include<limits>
00010 #include<iostream>
00011
00012
00013 /*
00014  My implementation of the Floyd_Warshall algorithm.
00015  Takes as input the adjacency matrix m of the graph.
00016  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.
00017  For every node i m[i][i] must be set to 0.
00018  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.
00019  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.
00020  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
00021  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)
00022  Time complexity: O(n^3)
00023 */

```

```

00024
00025 std::vector<std::vector<long long int>> Floyd_Warshall(std::vector<std::vector<long long int>>
    distance_matrix){
00026     const long long int inf = std::numeric_limits<long long int>::max() / 2;
00027     size_t n = distance_matrix.size();
00028
00029     for(size_t k = 0; k < n; k++){
00030         for(size_t i = 0; i < n; i++){
00031             for(size_t j = 0; j < n; j++){
00032                 if(distance_matrix[i][k] == inf || distance_matrix[k][j] == inf) continue;
00033                 distance_matrix[i][j] = std::min(distance_matrix[i][j], distance_matrix[i][k] +
    distance_matrix[k][j]);
00034             }
00035         }
00036     }
00037
00038
00039     for(size_t i = 0; i < n; i++){
00040         for(size_t j = 0; j < n; j++){
00041             for(size_t k = 0; k < n; k++){
00042                 if( distance_matrix[i][k] != inf &&
00043                    distance_matrix[k][j] != inf &&
00044                    distance_matrix[k][k] < 0){
00045                     distance_matrix[i][j] = -inf;
00046                 }
00047             }
00048         }
00049     }
00050
00051     return distance_matrix;
00052 }
00053
00054 #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.](#)

```

00001 /*
00002  AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef MAXIMUM_FLOW_HPP
00006 #define MAXIMUM_FLOW_HPP
00007
00008 #include<vector>
00009 #include<stddef.h>
00010 #include<queue>
00011 #include<optional>
00012
00013 struct Edge {
00014     size_t from; //Vilken nod går denna kanten från?
00015     size_t to; //Vilken nod går denna kanten till?
00016     long long int capacity; //Hur mycket kapacitet finns kvar?
00017     long long int flow = 0; //Hur mycket flöde är det just nu?
00018     long long int reverse = -1; //Var finns reverse-flow kanten?
00019 };
00020
00021 //Beräkna nivån för varje nod. D.v.s. hur långt ifrån källan den ligger. Om en nod inte kan nås från
00022 //källan då får den värdet -1
00023 bool dinic_bfs(const size_t source, const size_t sink, const std::vector<std::vector<Edge>& edges,
00024               std::vector<long long int>& level);
00025
00026 //Hitta vägar som expanderar flödet med en DFS.
00027 long long int dinic_dfs(const size_t at, const size_t sink, std::vector<std::vector<Edge>& edges,
00028                       const std::vector<long long int>& level, std::vector<size_t>& next, long long int mini);
00029
00030 /*
00031  My implementation of dinitz algorithm to calculate the maximum flow in a graph.
00032  Takes as input the number of nodes in the graph, what node is the source, what node is the sink, and
00033  the edges of the graph in adjacency list representation.
00034  Returns a pair, where the first element is the size of the maximum possible flow for the graph. The
00035  second element is a vector containing all edges that there is flow through.
00036  Time complexity: O(m*n^2) where n is the number of nodes in the graph, and m is the number of edges.
00037 */
00038 std::pair<long long int, std::vector<Edge>> dinic_max_flow(const size_t nodes, const size_t source,
00039                 const size_t sink, std::vector<std::vector<Edge>> edges);
00040
00041
00042
00043

```

```

00044
00045
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064
00065
00066
00067
00068
00069
00070 bool dinic_bfs(const size_t source, const size_t sink, const std::vector<std::vector<Edge>& edges,
std::vector<long long int>& level){
00071     for (size_t i = 0; i < level.size(); i++) {
00072         level[i] = -1;
00073     }
00074
00075     level[source] = 0;
00076
00077     std::queue<size_t> q;
00078     q.push(source);
00079     size_t at;
00080     while (!q.empty()) {
00081         at = q.front();
00082         q.pop();
00083
00084         for (size_t i = 0; i < edges[at].size(); i++) {
00085             if (level[edges[at][i].to] < 0 && edges[at][i].flow < edges[at][i].capacity) {
00086                 level[edges[at][i].to] = level[at] + 1;
00087                 q.push(edges[at][i].to);
00088             }
00089         }
00090     }
00091
00092     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.
00093 }
00094
00095 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){
00096     if (at == sink) return mini;
00097
00098     //Next sparas mellan olika anrop av dfs. D.v.s vi börjar där vi vet att det kan finnas utrymme att
expandera.
00099     for (; next[at] < edges[at].size(); next[at]++) {
00100         Edge& e = edges[at][next[at]];
00101
00102         //Om noden vi tittar på har en högre level och kanten har kapacitet.
00103         if (level[e.to] == level[at] + 1 && e.flow < e.capacity) {
00104
00105             //T1 är den minsta kapaciteten hos alla kanter som går till sänkan. D.v.s. maximala
möjliga utvidgninen.
00106             long long int t1 = dinic_dfs(e.to, sink, edges, level, next, std::min(mini, e.capacity -
e.flow));
00107             if (t1) {
00108                 e.flow += t1;
00109                 edges[e.to][e.reverse].flow -= t1;
00110                 return t1;
00111             }
00112         }
00113     }
00114     return 0;
00115 }
00116
00117 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){
00118     long long int flowIncrease;
00119     long long int total = 0;
00120
00121     //Add reverse flow edges
00122     for(size_t node = 0; node < nodes; node++){
00123         for(size_t i = 0; i < edges[node].size(); i++){

```

```

00124         Edge& e = edges[node][i];
00125         if (e.reverse == -1){
00126             e.reverse = edges[e.to].size();
00127             Edge reverse = {e.to, e.from, 0, 0, i};
00128             edges[e.to].push_back(reverse);
00129         }
00130         else{
00131             break;
00132         }
00133     }
00134 }
00135
00136 std::vector<long long int> level(nodes);
00137 std::vector<size_t> next(2*nodes);
00138 while (dinic_bfs(source, sink, edges, level)) {
00139     for (size_t i = 0; i < nodes; i++) {
00140         next[i] = 0;
00141     }
00142     do {
00143         flowIncrease = dinic_dfs(source, sink, edges, level, next, (1 « 30));
00144         total += flowIncrease;
00145     } while (flowIncrease); //Medans det finns flöde att expandera med, gör det.
00146 }
00147
00148 std::vector<Edge> flowEdges;
00149 for (size_t i = 0; i < nodes; i++) {
00150     for (size_t j = 0; j < edges[i].size(); j++) {
00151         if (edges[i][j].flow > 0) flowEdges.push_back(edges[i][j]);
00152     }
00153 }
00154 }
00155
00156 return {total, flowEdges};
00157 }
00158
00159
00160
00161
00162 #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.](#)

```

00001  /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef MINIMUM_CUT
00006  #define MINIMUM_CUT
00007
00008  #include "Graphs\Maximum_Flow\Maximum_Flow.hpp"
00009
00010  /*
00011  My implementation of using dinitz algorithm to calculate the minimum cut of a graph.
00012  Takes as input the number of nodes in the graph, the vertice s that should be in the cut, the vertice
00013  v which must not be in the cut. And the edges of the graph as adjacency lists
00014  Returns a vector containing all the nodes that are included in the cut.
00015  Time complexity: O(m*n^2) where n is the number of nodes in the graph, and m is the number of edges.
00016  */
00017  std::vector<size_t> minimum_cut(const size_t nodes, const size_t s, const size_t t,
00018                                std::vector<std::vector<Edge>> edges){
00019      long long int flowIncrease;
00020      long long int total = 0;
00021
00022      //Add reverse flow edges
00023      for(size_t node = 0; node < nodes; node++){
00024          for(size_t i = 0; i < edges[node].size(); i++){
00025              Edge& e = edges[node][i];
00026              if (e.reverse == -1){
00027                  e.reverse = edges[e.to].size();
00028                  Edge reverse = {e.to, e.from, 0, 0, i};
00029                  edges[e.to].push_back(reverse);
00030              }
00031              else{
00032                  break;
00033              }
00034          }
00035      }
00036
00037      std::vector<long long int> level(nodes);
00038      std::vector<size_t> next(2*nodes);
00039      while (dinic_bfs(s, t, edges, level)) {
00040          for (size_t i = 0; i < nodes; i++) {
00041              next[i] = 0;
00042          }
00043          do {
00044              flowIncrease = dinic_dfs(s, t, edges, level, next, (1 << 30));
00045              total += flowIncrease;
00046          } while (flowIncrease); //Medans det finns flöde att expandera med, gör det.
00047      }
00048
00049      //The nodes to be included in the cut are all nodes still reachable from s
00050      std::vector<size_t> minCutNodes;
00051      for (size_t i = 0; i < nodes; i++) {
00052          if(level[i] != -1) minCutNodes.push_back(i);
00053      }
00054
00055      return minCutNodes;
00056  }
00057
00058  #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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef MINIMIAL_SPANNING_TREE_HPP
00006 #define MINIMIAL_SPANNING_TREE_HPP
00007
00008 #include<vector>
00009 #include<utility>
00010 #include<algorithm>
00011 #include<stddef.h>
00012 #include"Data_Structures\Disjoint_Set\DisjointSet.hpp"
00013
00014 /*
00015 Function used to calculate the minimum spanning tree of a graph.
00016 Takes as input the number of vertices and a vector of edges.
00017 An edge is represented as a pair of 1. the edge weight w, and 2. a pair containing nodes u & v. This
00018 means there is an edge between u & v with weight w
00019 Returns a pair containing 1. the size of the tree, and 2. a vector containing pairs what nodes there
00020 are edges between.
00021 If there is no minimum spanning tree then the size is returned as -1 and the vector is empty.
00022 Time complexity: O(m * log(n)) where n is the number of nodes and m is the number of edges in the
00023 graph.
00024 */
00025 std::pair<long long int, std::vector<std::pair<size_t, size_t>> mst(size_t vertices,
00026 std::vector<std::pair<long long int, std::pair<size_t, size_t>> edges){
00027     long long int cost = 0;
00028     std::vector<std::pair<size_t,size_t> included;
00029     std::sort(edges.begin(), edges.end());
00030     DisjointSet s(vertices);
00031     for(auto a : edges) {
00032         if (!s.query(a.second.first, a.second.second)) {
00033             s.unionSets(a.second.first, a.second.second);
00034             included.push_back({a.second.first, a.second.second});
00035             cost += a.first;
00036         }
00037     }
00038     if(included.size() != vertices-1) return {-1, {}};
00039     return {cost,included};
00040 }
```

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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef EQUATION_SOLVER_HPP
00006 #define EQUATION_SOLVER_HPP
00007 #include<vector>
00008 #include<cmath>
00009 #include<iostream>
00010 #include<assert.h>
00011
00012
00013 //avoid silly precision errors
00014 bool is_double_zero(double x){
00015     if(x < 0) x = -x;
00016     return x < 0.000000001;
00017 }
00018
00019 /*
00020 Tar en n X n matris a och en vector b av längd n. Löser sedan ekvationssystemet ax = b med hjälp av
    gauss-eliminering
00021 och returnerar vektorn v av längd n där v[i] = värdet för x_i
00022 Om ett x_i inte kan bestämmas utan har flera möjliga värden då är v[i] = NaN.
00023 Om det linjära ekvationssystemet är inkonsistent då returneras en tom vektor.
00024 */
00025
00026 std::vector<double> equation_solver(const std::vector<std::vector<double>& a, const
    std::vector<double>& b){
00027     std::vector<std::vector<double>> m = a;
00028     std::vector<double> ans;
00029
00030     for(int row = 0; row < a.size(); row++){
00031         m[row].push_back(b[row]);
00032     }
00033
00034     int rowStart = 0;
00035
00036     //Triangulering
00037     for(int col = 0; col < m.size(); col++){
00038         int index = col;
00039
00040
00041         for(int row = rowStart; row < m.size(); row++){
00042             if(std::abs(m[row][col]) > std::abs(m[index][col])) index = row;
```



```

00043     }
00044
00045     if(is_double_zero(m[index][col])){
00046         continue;
00047     }
00048
00049
00050     m[rowStart].swap(m[index]);
00051
00052     for(int col2 = m[0].size()-1; col2 >= col; col2--){
00053         m[rowStart][col2] /= m[rowStart][col];
00054     }
00055
00056     for(int row = rowStart+1; row < m.size(); row++){
00057         if(is_double_zero(m[rowStart][col])) continue;
00058         double mult = m[row][col]/m[rowStart][col];
00059
00060         for(int col2 = col; col2 < m[row].size(); col2++){
00061             m[row][col2] -= m[rowStart][col2] * mult;
00062         }
00063     }
00064
00065     rowStart++;
00066 }
00067
00068 //Bakâsubstitution
00069 for(int row1 = m.size()-1; row1 > 0; row1--){
00070     int col=-1;
00071     for(int column = 0; column < m.size(); column++){
00072         if(m[row1][column] == 1){
00073             col = column;
00074             break;
00075         }
00076     }
00077     if(col == -1) continue;
00078
00079     for(int row2 = row1-1; row2 >= 0; row2--){
00080         if(is_double_zero(m[row1][col])) continue;
00081         double mult = m[row2][col]/m[row1][col];
00082
00083         for(int col = 0; col <= m.size(); col++){
00084             m[row2][col] -= m[row1][col]*mult;
00085         }
00086     }
00087 }
00088
00089 }
00090
00091 //Check for inconsistency
00092 for(int row = 0; row < m.size(); row++){
00093     bool allZero = true;
00094
00095     for(int col = 0; col < m.size(); col++){
00096         if(!is_double_zero(m[row][col])) allZero = false;
00097     }
00098
00099     if(allZero && !is_double_zero(m[row][m.size()])) return {};
00100 }
00101
00102 //Construct solution
00103 rowStart = 0;
00104 for(int col = 0; col < m.size(); col++){
00105     if(is_double_zero(m[rowStart][col])){
00106         ans.push_back(std::nan(""));
00107         continue;
00108     }
00109
00110     bool allZero = true;
00111
00112     for(int col2 = col+1; col2 < m.size(); col2++){
00113         if(!is_double_zero(m[rowStart][col2])) allZero = false;
00114     }
00115
00116     if(allZero == false) ans.push_back(std::nan(""));
00117     else ans.push_back(m[rowStart][m.size()]);
00118
00119     rowStart++;
00120 }
00121 assert(ans.size() == b.size());
00122 return ans;
00123 }
00124 }
00125
00126 #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.](#)

```
00001 /*
00002     AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef COVER_HPP
00006 #define COVER_HPP
00007
00008 #include<vector>
00009 #include<utility>
00010 #include<algorithm>
00011
00012 /*
00013 Takes an interval to be covered as a pair of doubles, with the first pair value being the left bound
00014 of the interval,
00015 and the second pair value being the right bound of the interval. It also takes a list of intervals
00016 that should be used to
00017 try and cover this interval. Returns a vector containing the indices in the intervals vector relating
00018 to the intervals that need to be used.
00019 If it is not possible to cover interval then an empty vector is returned.
00020 O(n log(n)) - where n is the size of the intervals vector.
00021
00022 TODO: Template it & convert intervals to be range based.
00023 */
00024
00025 std::vector<size_t> cover(const std::pair<double, double> interval, const
00026 std::vector<std::pair<double, double>& intervals){
00027
00028     //Tripple will be used later to store intervals (first, second) as well as their index in the
00029     original intervals array.
00030     struct Tripple{
00031         double first, second;
00032         size_t third;
00033     };
00034
00035     double l = interval.first, r = interval.second;
00036     if(l == r){ //If the interval is a single point then deal with that case
00037         for(size_t i = 0; i < intervals.size(); i++){
00038             if(intervals[i].first <= l && intervals[i].second >= r) return {i};
00039         }
00040     }
00041     return {};
```

```

00038     }
00039
00040
00041     //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.
00042     std::vector<Tripple> intervalsCopy;
00043
00044     intervalsCopy.reserve(intervals.size());
00045
00046     for(size_t i = 0; i < intervals.size(); i++){
00047         intervalsCopy.push_back({intervals[i].first, intervals[i].second, i});
00048     }
00049
00050     std::sort(intervalsCopy.begin(), intervalsCopy.end(), [](Tripple& a, Tripple& b){
00051         return a.first < b.first;
00052     });
00053
00054     size_t index = 0;
00055     size_t bestIndex = 0;
00056     double reach = 1;
00057
00058     std::vector<size_t> solution;
00059
00060
00061     while(l < r){
00062         while(index < intervalsCopy.size() && intervalsCopy[index].first <= l){
00063
00064             //Find whatever interval that begins before current l that reaches the furthest.
00065             if(intervalsCopy[index].second > reach){
00066                 bestIndex = intervalsCopy[index].third;
00067                 reach = intervalsCopy[index].second;
00068             }
00069             index++;
00070         }
00071
00072         //If we couldn't extend the interval we are working on then return empty vector.
00073         if(reach == l) return {};
00074
00075         l = reach;
00076         solution.push_back(bestIndex);
00077     }
00078
00079     return solution;
00080 }
00081
00082
00083 #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.](#)

```

00001  /*
00002      AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef KNAPSACK_HPP
00006  #define KNAPSACK_HPP
00007  #include<vector>
00008  #include<utility>
00009  #include<algorithm>
00010
00011  /*
00012  A function that solves the knapsack problem and returns the indices in the items vector that should be
00013  used to obtain the optimal value.
00014  Capacity is the maximum weight that the knapsack can hold. Must be non-negative.
00015  Items is a list of non-negative integer pairs where the first value in each pair is the weight of the
00016  item, and the second pair is the value.
00017  O(n*k) - where n is the size of the items vector and k is the capacity
00018  TODO: Change to iterator based implementation, make capacity size_t, template it.
00019  */
00020  std::vector<size_t> knapsack(const int capacity, const std::vector<std::pair<int, int>>& items){
00021      std::vector<size_t> indices;
00022
00023      std::vector<std::vector<int>> dp(capacity+1);
00024      for(size_t i = 0; i <= capacity; i++){
00025          dp[i].resize(items.size()+1);
00026      }
00027
00028      //Fill out DP
00029      for(int weight = capacity; weight >= 0; --weight){
00030          for(int item = items.size(); item >= 0; --item){
00031              if(weight == capacity || item == items.size())
00032                  dp[weight][item] = 0;
00033              else if (weight+items[item].first <= capacity)
00034                  dp[weight][item] = std::max(dp[weight][item + 1], dp[weight+items[item].first][item +
00035  1] + items[item].second);
00036              else
00037                  dp[weight][item] = dp[weight][item + 1];
00038          }
00039      }
00040
00041      //Reconstruct the solution
00042      int weight = 0;
00043      size_t item = 0;
00044      while(item != items.size() && weight != capacity){
00045          if (weight+items[item].first <= capacity && dp[weight+items[item].first][item + 1] +
00046              items[item].second > dp[weight][item + 1]){
00047              indices.push_back(item);
00048              weight += items[item].first;
00049              item++;
00050          }
00051          else{
00052              item++;
00053          }
00054      }
00055      return indices;
00056  }
00057
00058  #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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003 */
00004
00005 #ifndef LONGEST_INCREASING_SUBSEQUENCE_HPP
00006 #define LONGEST_INCREASING_SUBSEQUENCE_HPP
00007 #include<vector>
00008 #include<iterator>
00009 #include<limits>
00010 #include<algorithm>
00011
00012 /*
00013  Takes a vector of integers and returns a vector of the indicies that should be used if you want to
00014  construct the longest possible subsequence using those integers.
00015  O(N*log(N))
00016 */
00017 std::vector<size_t> lis(const std::vector<long long int>& v){
00018     typedef long long int ll;
00019     const ll inf = std::numeric_limits<ll>::max();
00020
00021     std::vector<size_t> ans;
00022     std::vector<std::pair<ll, size_t>> last(v.size()+1, {inf, inf}); //Last[L].first is the lowest
00023     value so far that can end a subsequence of length L. Second är L's index i v
00024     std::vector<size_t> backTrack(v.size(), inf);
00025     last[0].first = -last[0].first;
00026
00027     for(size_t i = 0; i < v.size(); i++){
00028         //it pekar på första elementet som är större än v[i].
00029         auto it = std::upper_bound(last.begin(), last.end(), v[i], [](auto value, auto elem){
00030             return value < elem.first;
00031         });
00032
00033         if((it-1)->first < v[i]){ //Kolla så att det går att avsluta en subsequence av längd L-1 med
00034             ett värde som är mindre än v[i]
00035             it->first = v[i];
00036             it->second = i;
00037             backTrack[i] = (it-1)->second;
00038         }
00039     }
00040
00041     size_t L;
00042     for(size_t i = 1; i < last.size(); i++){
00043         if(last[i].first < inf){
00044             L = last[i].second;
00045         }
00046     }
00047
00048     //Reconstruct the solution
00049     ans.reserve(L);
00050     do{
00051         ans.push_back(L);
00052         L = backTrack[L];
00053     }while(L != inf);
00054
00055     std::reverse(ans.begin(), ans.end());
00056
00057     return ans;
00058 }
00059
00060 #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.](#)

```
00001 /*
00002 AUTHOR: Oliver Lindgren
00003 */
00004
00005
00006 #ifndef FFT_HPP
00007 #define FFT_HPP
00008
00009 #define PI 3.14159265359
00010
00011 #include<vector>
00012 #include<complex>
00013
00014 typedef long long int ll;
00015 /*
00016 Takes two polynomials in coefficient list representation, where the coefficients are whole integers,
00017 multiplies them together using the fast fourier transformation algorithm
00018 and returns the resulting polynomial also in coefficient list representation.
00019 O(N log(N))
00020 */
00021 std::vector<ll> Polymul(std::vector<ll> a, std::vector<ll> b);
00022
00023 /*
00024 Takes two polynomials in coefficient list representation, where the coefficients are whole integers,
00025 multiplies them together using a naive O(N^2) algorithm
00026 and returns the resulting polynomial also in coefficient list representation.
00027 O(N^2)
00028 */
00029 std::vector<ll> Polymul_Quadratic(std::vector<ll> a, std::vector<ll> b);
00030
00031 /*
00032 Applies the FFT on the vector P and returns the result. N is the size of the vector P.
00033 Start and skip are used to index the vector and should be left default initialized.
00034 O(N log(N))
00035 */
00036 std::vector<std::complex<double>> FFT(const std::vector<ll>& P, int n, int start = 0, int skip = 1);
00037
00038 /*
00039 Applies the inverse FFT on the vector P and returns the result. N is the size of the vector P.
00040 Start and skip are used to index the vector and should be left default initialized.
00041 O(N log(N))
00042 */
```

```

00043 std::vector<std::complex<double>> INVERSE_FFT(const std::vector<std::complex<double>>& P, int n, int
      start = 0, int skip = 1);
00044
00045
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064
00065
00066
00067
00068
00069
00070
00071
00072
00073
00074 std::vector<ll> Polymul(std::vector<ll> a, std::vector<ll> b){
00075     int n=1;
00076     int org = a.size() + b.size() - 1;
00077
00078     while(n < org) n*=2;
00079
00080     while(a.size() < n) a.push_back(0);
00081     while(b.size() < n) b.push_back(0);
00082
00083     std::vector<std::complex<double>> a_value_rep = FFT(a, n);
00084     std::vector<std::complex<double>> b_value_rep = FFT(b, n);
00085     std::vector<std::complex<double>> merged;
00086     std::vector<std::complex<double>> reversed;
00087
00088     merged.reserve(n);
00089
00090     for(int i = 0; i < n; i++){
00091         merged.push_back(a_value_rep[i] * b_value_rep[i]);
00092     }
00093
00094     reversed = INVERSE_FFT(merged, n);
00095
00096     std::vector<ll> ans;
00097     ans.reserve(org);
00098
00099     for(int i = 0; i < org; i++){
00100         ans.push_back(round(reversed[i].real()/(double)n));
00101     }
00102
00103     return ans;
00104 }
00105
00106
00107 std::vector<std::complex<double>> FFT(const std::vector<ll>& p, int n, int start, int skip){
00108     using namespace std::complex_literals;
00109
00110     if(p.size() == skip){
00111         return {(double)p[start], 0.0};
00112     } //Ett polynom av grad 0 har alltid värdet av koefficienten. FIIIIIIIX
00113
00114
00115
00116     std::vector<std::complex<double>> y_even, y_odd;
00117     y_even = FFT(p, n/2, start, skip*2);
00118     y_odd = FFT(p, n/2, start+skip, skip*2);
00119
00120     std::vector<std::complex<double>> y_merged(n);
00121     for(int i = 0; i < n/2; i++){
00122         std::complex<double> compute = std::exp((2.0*PI*1i*(double)i)/(double)n) * y_odd[i];
00123         y_merged[i] = y_even[i] + compute;
00124         y_merged[i+n/2] = y_even[i] - compute;
00125     }
00126     return y_merged;
00127 }
00128

```



```

00129
00130 std::vector<std::complex<double>> INVERSE_FFT(const std::vector<std::complex<double>>& p, int n, int
    start, int skip){
00131     using namespace std::complex_literals;
00132
00133     if(p.size() == skip){
00134         return {p[start]};
00135     } //Ett polynom av grad 0 har alltid värdet av koefficienten. FIIIIIIIX
00136
00137     std::vector<std::complex<double>> y_even, y_odd;
00138     y_even = INVERSE_FFT(p, n/2, start, skip*2);
00139     y_odd = INVERSE_FFT(p, n/2, start+skip, skip*2);
00140
00141     std::vector<std::complex<double>> y_merged(n);
00142     for(int i = 0; i < n/2; i++){
00143         std::complex<double> compute = std::exp((-2.0*PI*1i*(double)i)/(double)n) * y_odd[i];
00144         y_merged[i] = y_even[i] + compute;
00145         y_merged[i+n/2] = y_even[i] - compute;
00146     }
00147     return y_merged;
00148 }
00149
00150 std::vector<ll> Polymul_Quadratic(std::vector<ll> a, std::vector<ll> b){
00151     std::vector<ll> res(a.size() + b.size() - 1);
00152
00153     for(int i = 0; i < a.size(); i++){
00154         for(int j = 0; j < b.size(); j++){
00155             res[i+j] += a[i]*b[j];
00156         }
00157     }
00158
00159     return res;
00160 }
00161
00162 #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.](#)

```
00001 /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005
00006 #include<cmath>
00007 #include<vector>
00008
00009 #include"Prime_Numbers\Prime_Sieve\Sieve.hpp"
00010
00011 using namespace std;
00012
00013 typedef long long int ll;
00014 typedef unsigned long long int ull;
00015 typedef __uint128_t u128;
00016
00017 vector<size_t> primes;
00018
00019 inline ull modMult(const ull a, const ull b, const ull m){
00020     return (u128)a * b % m;
00021 }
00022
00023 ull powMod(ull base, ull exp, ull m) {
00024     ull result = 1;
00025     base %= m;
00026     while (exp) {
00027         if (exp % 2 == 1)
00028             result = (u128)result * base % m;
00029         base = (u128)base * base % m;
00030         exp /= 2;
00031     }
00032     return result;
00033 }
00034
00035 bool Miller_test(ull n, ull a, ull d, int s) {
00036     ull x = powMod(a, d, n);
00037     if (x == 1 || x == n - 1)
00038         return false;
00039     for (int r = 1; r < s; r++) {
00040         x = modMult(x, x, n);
00041         if (x == n - 1)
00042             return false;
00043     }
00044     return true;
00045 };
00046
00047 bool is_Probably_Prime(ull n, int k=5) { // returns true if n is probably prime, else returns false.
00048     if (n == 1 || n == 4)
00049         return false;
00050     if (n <= 5) return true;
00051
00052     int s = 0;
00053     ull d = n - 1;
00054     while (d % 2 == 0) {
00055         d /= 2;
00056         s++;
00057     }
00058
00059     for (int i = 0; i < k; i++) {
00060         int a = 2 + rand() % (n - 3);
00061         if (Miller_test(n, a, d, s))
00062             return false;
00063     }
00064     return true;
00065 }
00066
00067 ll gcd(ll a, ll b){
00068     a = std::abs(a);
```

```

00069     b = std::abs(b);
00070     if(a > b){
00071         std::swap(a,b);
00072     }
00073
00074     if (a == 0)
00075         return b;
00076     return gcd(b % a, a);
00077 }
00078
00079 inline ull f(ull x, ull c, ull m){
00080     return (modMult(x, x, m) + c) % m;
00081 }
00082
00083 ull Pollard_Rho(ull n, ull c = 1, ull x0 = 2){
00084     ll hare = x0;
00085     ll tortoise = x0;
00086     ull p = 1;
00087
00088     ull t = 0;
00089     while(p == 1){
00090         t++;
00091         tortoise = f(tortoise, c, n);
00092         hare = f(hare, c, n);
00093         hare = f(hare, c, n);
00094
00095         p = gcd(abs(tortoise-hare), n);
00096     }
00097
00098     if(p == n) return Pollard_Rho(n, c+1, x0);
00099     else return p;
00100 }
00101 }
00102
00103 //Almost identical to fast_prime except it keeps track of how many times a prime appears
00104 vector<pair<ull,ull> prime_with_mult(ull n){
00105     vector<pair<ull,ull> ans;
00106
00107     for(auto x : primes){
00108         if(n % x == 0) ans.push_back({x, 0});
00109         while(n % x == 0){
00110             n /= x;
00111             ans.rbegin()->second++;
00112         }
00113
00114         if(x*x > n) break;
00115     }
00116
00117     if(n == 1) return ans;
00118
00119     if(!is_Probably_Prime(n, 10)){
00120         ull p = Pollard_Rho(n);
00121         if(p*p == n){
00122             ans.push_back({p,2});
00123         }
00124         else{
00125             ans.push_back({p, 1});
00126             ans.push_back({n/p, 1});
00127         }
00128     }
00129     else{
00130         ans.push_back({n, 1});
00131     }
00132     return ans;
00133 }
00134
00135 vector<ull> fast_prime(ull n){
00136     vector<ull> ans;
00137
00138     for(auto x : primes){
00139         if(n % x == 0) ans.push_back(x);
00140         while(n % x == 0) n /= x;
00141
00142         if(x*x > n) break;
00143     }
00144
00145     if(n == 1) return ans;
00146
00147     if(!is_Probably_Prime(n, 10)){
00148         ull p = Pollard_Rho(n);
00149
00150         if(p*p == n){
00151             ans.push_back(p);
00152         }
00153         else{
00154             ans.push_back(p);
00155             ans.push_back(n/p);

```

```

00156     }
00157 }
00158 else{
00159     ans.push_back(n);
00160 }
00161 return ans;
00162 }
00163
00164 void nested(vector<pair<ull,ull>& v, const vector<pair<ull,ull>& org, int depth, vector<ull>& ans){
00165     if(depth == v.size()){
00166         ull d = 1;
00167         for(int i = 0; i < v.size(); i++){
00168             d *= pow(v[i].first, v[i].second);
00169         }
00170         ans.push_back(d);
00171         return;
00172     }
00173
00174     while(v[depth].second >= 0){
00175         nested(v, org, depth+1, ans);
00176
00177         if(v[depth].second == 0)break;
00178         v[depth].second--;
00179     }
00180     v[depth].second = org[depth].second;
00181     return;
00182 }
00183
00184
00185 vector<ull> fast_factors(ull n){
00186     Prime_Sieve sieve(4e4 + 7e3); //Ty 14/3 ~= 4.7
00187     primes = sieve.get_Primes();
00188
00189
00190     auto v = prime_with_mult(n);
00191     auto org = v;
00192     vector<ull> ans;
00193
00194     nested(v, org, 0, ans);
00195     return ans;
00196 }

```

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.](#)

```

00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef PRIME_SIEVE_HPP
00006 #define PRIME_SIEVE_HPP
00007
00008 #include<vector>
00009 #include<stddef.h>
00010 #include<iostream>
00011
00012 /*
00013     Class to compute all primes up to a number N. it provides the following functions:
00014

```

```

00015     Prime_Sieve(N) - the constructor takes a number N as an argument and computes which numbers up to
N that are prime.
00016     get_Number_Of_Primes() - returns the number of primes that are smaller than or equal to N.
00017     get_Primes() returns a sorted vector of all numbers less than N that are prime.
00018     is_Prime(x) - returns true if x is prime. Returns false otherwise.
00019 */
00020
00021 class Prime_Sieve{
00022     std::vector<bool> is_composite;
00023     std::vector<size_t> primes;
00024
00025     public:
00026     const std::vector<size_t>& get_Primes();
00027     Prime_Sieve(size_t n);
00028     size_t get_Number_Of_Primes();
00029     bool is_Prime(size_t i);
00030 };
00031
00032
00033
00034
00035
00036
00037
00038
00039
00040
00041
00042
00043
00044
00045
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064
00065
00066
00067 Prime_Sieve::Prime_Sieve(size_t n) : is_composite(n+1){
00068     n++;
00069     is_composite[0] = true;          //0 is not a composite number, however, it is not prime
either.
00070     if(n > 1) is_composite[1] = true; //The same applies to the number 1.
00071     if(n > 2) primes.push_back(2);
00072
00073
00074     size_t x = 4;
00075     while(x < n){
00076         is_composite[x] = true;
00077         x+=2;
00078     }
00079
00080
00081     for(size_t i = 3; i < n; i+=2){
00082         if(is_composite[i]) continue;
00083
00084         primes.push_back(i);
00085
00086         size_t mult = i*i;
00087         size_t add = 2*i;
00088         while(mult < n){
00089             is_composite[mult] = true;
00090             mult += add;
00091         }
00092
00093     }
00094
00095 }
00096
00097 bool Prime_Sieve::is_Prime(size_t i){
00098     return !is_composite[i];
00099 }

```

```

00100
00101 size_t PrimeSieve::get_Number_Of_Primes(){
00102     return primes.size();
00103 }
00104
00105 const std::vector<size_t>& PrimeSieve::get_Primes(){
00106     return primes;
00107 }
00108
00109 #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.](#)

```

00001 /*
00002 Author: Oliver Lindgren
00003 */
00004
00005 #ifndef AHO_CORASICK_HPP
00006 #define AHO_CORASICK_HPP
00007 #include<vector>
00008 #include<string>
00009 #include<queue>
00010 #include<iostream>
00011
00012 /*
00013 A class that implements the Aho-Corasick string matching algorithm.
00014 It provides three functions of interest:
00015
00016 add_string(s) adds a string pattern to be matched against.
00017 construct_automaton() constructs the automaton that will be used in the string matching. Call this
00018 once all strings you want to match against have been added.
00019 search(s) searches through the string s. Returns a vector V of vectors containing the indicies in s
00020 where a pattern was sucessfully matched. V[i] contains the positions of where the ith string added
00021 match.
00022 */
00023
00024 class TrieAutomaton{
00025 struct Node{
00026     std::vector<int> terminal;
00027     int depth;
00028     Node* suffixNode = nullptr;
00029     Node* terminalChain;
00030     char nodeChar = 'X';
00031
00032     std::vector<Node*> transition = std::vector<Node*>(128, nullptr);
00033 };
00034
00035 Node* root = new Node();
00036 size_t stringIndex = 0;
00037 std::vector<Node*> nodes;
00038

```



```

00036 public:
00037 TrieAutomaton();
00038 ~TrieAutomaton();
00039
00040 void add_string(std::string const & s);
00041 void construct_automaton();
00042
00043 std::vector<std::vector<size_t>> search(std::string const & s);
00044
00045 };
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059
00060
00061
00062
00063
00064
00065
00066
00067
00068
00069
00070
00071
00072
00073
00074
00075
00076
00077
00078
00079
00080
00081
00082
00083
00084
00085
00086
00087
00088
00089
00090
00091
00092
00093
00094 void TrieAutomaton::add_string(std::string const & s){
00095     Node* at = root;
00096
00097     for(char c : s){
00098         if(at->transition[c] == nullptr){
00099             Node* newNode = new Node();
00100             nodes.push_back(newNode);
00101             newNode->depth = at->depth + 1;
00102             newNode->nodeChar = c;
00103             at->transition[c] = newNode;
00104         }
00105         at = at->transition[c];
00106     }
00107
00108     at->terminal.push_back(stringIndex);
00109     stringIndex++;
00110 }
00111
00112 void TrieAutomaton::construct_automaton(){
00113     Node* at;
00114     std::queue<Node*> bfsQ;
00115     bfsQ.push(root);
00116
00117     while(!bfsQ.empty()){
00118         at = bfsQ.front();
00119         bfsQ.pop();
00120         for(unsigned char c = 0; c < 128; c++){
00121             if(at->transition[c] == nullptr){
00122                 if(at == root) at->transition[c] = root;

```

```

00123         else at->transition[c] = at->suffixNode->transition[c];
00124     }
00125 }
00126 else{
00127     if(at == root) at->transition[c]->suffixNode = root;
00128     else at->transition[c]->suffixNode = at->suffixNode->transition[c]; //Could probably
make some chain here during construction.
00129
00130 //Suffix chain
00131 if(!at->suffixNode->transition[c]->terminal.empty()){
00132     at->transition[c]->terminalChain = at->suffixNode->transition[c];
00133 }
00134 else{
00135     at->transition[c]->terminalChain = at->suffixNode->transition[c]->terminalChain;
00136 }
00137
00138 if(at == root){
00139     at->transition[c]->terminalChain = root;
00140 }
00141     bfsQ.push(at->transition[c]);
00142 }
00143 }
00144 }
00145 }
00146
00147 std::vector<std::vector<size_t> TrieAutomaton::search(std::string const & s){
00148     Node* at = root;
00149     std::vector<std::vector<size_t> ans(stringIndex);
00150     for(size_t i = 0; i < s.length(); i++){
00151         char c = s[i];
00152         at = at->transition[c];
00153
00154         Node* suffixLinking = at;
00155         Node* prev = nullptr;
00156         while(suffixLinking != root){ //Somehow save the previous terminal suffix for all nodes to
save time.
00157             if(suffixLinking == prev) std::cerr << "Linking err";
00158
00159             for(auto index : suffixLinking->terminal){
00160                 ans[index].push_back(i - suffixLinking->depth + 1);
00161             }
00162
00163             prev = suffixLinking;
00164             suffixLinking = suffixLinking->terminalChain;
00165         }
00166     }
00167     return ans;
00168 }
00169
00170 TrieAutomaton::TrieAutomaton(){
00171     root->suffixNode = root;
00172     root->depth = 0;
00173     root->terminalChain = root;
00174 }
00175
00176 TrieAutomaton::~TrieAutomaton(){
00177     delete root;
00178     for(auto x : nodes){
00179         delete x;
00180     }
00181 }
00182 #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.](#)

```

00001  /*
00002  AUTHOR: Oliver Lindgren
00003  */
00004
00005  #ifndef SUFFIX_ARRAY_HPP
00006  #define SUFFIX_ARRAY_HPP
00007
00008  #include<string>
00009  #include<algorithm>
00010  #include<vector>
00011  #include<iostream>
00012  #include<stddef.h>
00013
00014  /*
00015  A class that computes the suffix-array of a string s.
00016  Takes a string to compute the array for as argument to the constructor.
00017  Also provides a function getSuffix(i) which returns the index in s at which the i:th smallest suffix
00018  begins.
00019  Time complexity of the constructor is O(n log^2(n)),
00020  getSuffix runs in constant time.
00021  */
00022  class SuffixArray{
00023  private:
00024      std::vector<size_t> sorted;
00025
00026  public:
00027      SuffixArray(std::string const & s);
00028      size_t getSuffix(size_t i);
00029  };
00030
00031
00032
00033
00034
00035
00036
00037
00038
00039
00040
00041
00042
00043
00044
00045
00046
00047
00048
00049
00050
00051
00052
00053
00054
00055
00056
00057
00058
00059  SuffixArray::SuffixArray(std::string const & s){
00060      std::vector<size_t> buckets(s.length());
00061      std::vector<size_t> newBuckets(s.length());
00062      std::vector<size_t> suffixPosition(s.length());
00063
00064      buckets[0] = 0;
00065      newBuckets[0] = 0;
00066
00067      for(size_t i = 0; i < s.size(); i++){
00068          sorted.push_back(i);
00069      }
00070
00071      //Do an initial sorting on just the first character
00072      std::sort(sorted.begin(), sorted.end(), [&s](const size_t a, const size_t b){
00073          return s[a] < s[b];
00074      });
00075
00076      buckets[0] = 0;
00077      suffixPosition[sorted[0]] = 0;
00078
00079      for(int i = 1; i < s.length(); i++){
00080          if(s[sorted[i]] == s[sorted[i-1]]) buckets[i] = buckets[i-1];
00081          else buckets[i] = buckets[i-1] + 1;

```

```

00082
00083     suffixPosition[sorted[i]] = buckets[i];
00084 }
00085
00086
00087
00088 //This loop will run log(n) times
00089 for(int k = 1; k <= s.length(); k *= 2){
00090
00091     //Sorting takes n log(n) time
00092     std::sort(sorted.begin(), sorted.end(), [&s, &k, &suffixPosition](const size_t a, const size_t
00093 b){
00094         if(suffixPosition[a] == suffixPosition[b]){
00095             //If one of the suffixes in this buckets end then it should come before the others
00096             if(a+k >= s.length() || b+k >= s.length()){
00097                 return a > b; // '>' instead of '<' because the bigger the number, the shorter the
00098 suffix
00099             }
00100             //Otherwise we sort based on the next 2^k chars, which have already been sorted
00101 somewhere else.
00102             return suffixPosition[a+k] < suffixPosition[b+k];
00103         }
00104         //If they don't belong to the same bucket then their relative position shall remain the
00105 same.
00106         return suffixPosition[a] < suffixPosition[b];
00107     });
00108
00109 //This will run n times for bookkeeping
00110 for(int i = 1; i < s.length(); i++){
00111     if(buckets[i] > buckets[i-1]) newBuckets[i] = newBuckets[i-1] + 1;
00112     else if(suffixPosition[sorted[i] + k] > suffixPosition[sorted[i-1] + k]) newBuckets[i] =
00113 newBuckets[i-1] + 1;
00114     else newBuckets[i] = newBuckets[i-1];
00115 }
00116 buckets.swap(newBuckets);
00117
00118 for(int i = 0; i < s.length(); i++){
00119     suffixPosition[sorted[i]] = buckets[i];
00120 }
00121 }
00122 size_t SuffixArray::getSuffix(size_t i){
00123     return sorted[i];
00124 }
00125 #endif

```

Index

- ~FenwickTree
 - FenwickTree, [10](#)
- ~TrieAutomaton
 - TrieAutomaton, [16](#)
- add
 - Points.hpp, [32](#)
- add_string
 - TrieAutomaton, [16](#)
- Algorithm_Library, [1](#)
- angle
 - Points.hpp, [32](#)
- Area.hpp
 - Polygon_area, [37](#)
- Arithmetic/Chinese-Remainder-Theorem/Chinese_Remainder.hpp,
[17](#), [18](#)
- Arithmetic/Modular-Arithmetic/modular.hpp, [18](#), [20](#)
- Arithmetic/Rational arithmetic/rational.hpp, [21](#), [22](#)
- Bellman_Ford
 - Bellman_Ford.hpp, [38](#)
- Bellman_Ford.hpp
 - Bellman_Ford, [38](#)
 - inf, [38](#)
 - negInf, [38](#)
- capacity
 - Edge, [8](#)
- Chinese_Remainder.hpp
 - Solve_System_Of_Congruencies, [17](#)
- construct_automaton
 - TrieAutomaton, [16](#)
- convex_hull
 - Convex_Hull.hpp, [27](#)
- Convex_Hull.hpp
 - convex_hull, [27](#)
- cover
 - cover.hpp, [56](#)
- cover.hpp
 - cover, [56](#)
- cross_product
 - Points.hpp, [33](#)
- Data_Structures/Disjoint_Set/DisjointSet.hpp, [23](#), [24](#)
- Data_Structures/FenwickTree/FenwickTree.hpp, [25](#)
- Data_Structures/Segment_Tree/Segment_Tree.hpp, [26](#)
- denominator
 - Rational < T >, [14](#)
- destination
 - Edge, [8](#)
- dijkstra_shortest_path
 - shortest_path.hpp, [40](#)
- dijkstra_shortest_path_time_table
 - shortest_path_time_table.hpp, [43](#)
- dinic_bfs
 - Maximum_Flow.hpp, [48](#)
- dinic_dfs
 - Maximum_Flow.hpp, [48](#)
- dinic_max_flow
 - Maximum_Flow.hpp, [49](#)
- DisjointSet, [7](#)
 - DisjointSet, [7](#)
 - query, [7](#)
 - unionSets, [7](#)
- dot_product
 - Points.hpp, [33](#)
- distance
 - shortest_path.hpp, [40](#)
 - shortest_path_time_table.hpp, [42](#)
- distSquared
 - Points.hpp, [33](#)
- dot_product
 - Points.hpp, [33](#)
- Edge, [8](#)
 - capacity, [8](#)
 - destination, [8](#)
 - firstDeparture, [8](#)
 - flow, [8](#)
 - from, [8](#)
 - repeatDepartures, [9](#)
 - reverse, [9](#)
 - to, [9](#)
 - travelTime, [9](#)
 - weight, [9](#)
- equation_solver
 - solver.hpp, [54](#)
- Eulerian_Path.hpp
 - Eulerian_Path_Directed, [44](#)
 - Eulerian_Path_Undirected, [44](#)
- Eulerian_Path_Directed
 - Eulerian_Path.hpp, [44](#)
- Eulerian_Path_Undirected
 - Eulerian_Path.hpp, [44](#)
- f
 - Factorize.hpp, [64](#)
- Factorize.hpp
 - f, [64](#)
 - fast_factors, [64](#)

- fast_prime, 64
- gcd, 64
- is_Probably_Prime, 64
- ll, 64
- Miller_test, 65
- modMult, 65
- nested, 65
- Pollard_Rho, 65
- powMod, 65
- prime_with_mult, 65
- primes, 66
- u128, 64
- ull, 64
- fast_factors
 - Factorize.hpp, 64
- fast_prime
 - Factorize.hpp, 64
- FenwickTree, 9
 - ~FenwickTree, 10
 - FenwickTree, 10
 - Query, 10
 - Update, 10
- FFT
 - FFT.hpp, 60
- FFT.hpp
 - FFT, 60
 - INVERSE_FFT, 60
 - ll, 60
 - PI, 60
 - Polymul, 61
 - Polymul_Quadratic, 61
- firstDeparture
 - Edge, 8
- flow
 - Edge, 8
- Floyd_Warshall
 - floyd_warshall.hpp, 47
- floyd_warshall.hpp
 - Floyd_Warshall, 47
- from
 - Edge, 8
- gcd
 - Factorize.hpp, 64
 - modular.hpp, 19
 - rational.hpp, 22
- Geometry/Convex_Hull/Convex_Hull.hpp, 27, 28
- Geometry/Inside_Polygon/Inside.hpp, 28, 30
- Geometry/Line_Intersection/intersection.hpp, 31
- Geometry/Points/Points.hpp, 32, 35
- Geometry/Polygon_Area/Area.hpp, 36, 37
- get_Number_Of_Primes
 - Prime_Sieve, 12
- get_Primes
 - Prime_Sieve, 12
- getSuffix
 - SuffixArray, 15
- Graphs/Bellman_Ford/Bellman_Ford.hpp, 37, 38
- Graphs/Dijkstra/shortest_path.hpp, 39, 41
- Graphs/Dijkstra_Time_Table/shortest_path_time_table.hpp, 41, 43
- Graphs/Eulerian_Path/Eulerian_Path.hpp, 44, 45
- Graphs/Floyd_Warshall/floyd_warshall.hpp, 47
- Graphs/Maximum_Flow/Maximum_Flow.hpp, 48, 49
- Graphs/Minimum_Cut/Minimum_Cut.hpp, 51, 52
- Graphs/MST/MST.hpp, 52, 53
- i128
 - modular.hpp, 19
- inf
 - Bellman_Ford.hpp, 38
- Inside
 - Inside.hpp, 29
- Inside.hpp
 - Inside, 29
 - inside_poly, 29
 - myAbs, 29
 - OnEdge, 29
 - onSegment, 29
 - Outside, 29
 - Status, 29
- inside_poly
 - Inside.hpp, 29
- intersect
 - Points.hpp, 33
- intersection.hpp
 - line_intersection, 31
- INVERSE_FFT
 - FFT.hpp, 60
- is_double_zero
 - solver.hpp, 54
- is_Prime
 - Prime_Sieve, 13
- is_Probably_Prime
 - Factorize.hpp, 64
- knapsack
 - Knapsack.hpp, 57
- Knapsack.hpp
 - knapsack, 57
- length
 - Point< T >, 11
 - Points.hpp, 33
- lengthSquared
 - Point< T >, 11
 - Points.hpp, 33
- line_intersection
 - intersection.hpp, 31
- lis
 - lis.hpp, 59
- lis.hpp
 - lis, 59
- ll
 - Factorize.hpp, 64
 - FFT.hpp, 60
 - modular.hpp, 19

- magnitude
 - Point< T >, 11
- magnitudeSquared
 - Point< T >, 11
- Maximum_Flow.hpp
 - dinic_bfs, 48
 - dinic_dfs, 48
 - dinic_max_flow, 49
- Miller_test
 - Factorize.hpp, 65
- minimum_cut
 - Minimum_Cut.hpp, 51
- Minimum_Cut.hpp
 - minimum_cut, 51
- Misc/Equation solver plus/solver.hpp, 53, 54
- Misc/Interval Cover/cover.hpp, 56
- Misc/Knapsack/Knapsack.hpp, 57, 58
- Misc/Longest Increasing Subsequence/lis.hpp, 58, 59
- Misc/Polymul/FFT.hpp, 60, 61
- mod
 - modular.hpp, 19
- modAdd
 - modular.hpp, 19
- modDiv
 - modular.hpp, 19
- modInverse
 - modular.hpp, 20
- modMult
 - Factorize.hpp, 65
 - modular.hpp, 20
- modSub
 - modular.hpp, 20
- modular.hpp
 - gcd, 19
 - i128, 19
 - ll, 19
 - mod, 19
 - modAdd, 19
 - modDiv, 19
 - modInverse, 20
 - modMult, 20
 - modSub, 20
 - u128, 19
 - ull, 19
- mst
 - MST.hpp, 53
- MST.hpp
 - mst, 53
- myAbs
 - Inside.hpp, 29
- negInf
 - Bellman_Ford.hpp, 38
- nested
 - Factorize.hpp, 65
- node
 - shortest_path.hpp, 40
 - shortest_path_time_table.hpp, 42
- normalise
 - Rational< T >, 13
- numerator
 - Rational< T >, 14
- OnEdge
 - Inside.hpp, 29
- onSegment
 - Inside.hpp, 29
- operator!=
 - Point< T >, 11
- operator<<
 - Points.hpp, 34
 - rational.hpp, 22
- operator>>
 - Points.hpp, 34
 - rational.hpp, 22
- operator+
 - Point< T >, 11
 - Rational< T >, 13
- operator-
 - Point< T >, 11
 - Rational< T >, 14
- operator/
 - Rational< T >, 14
- operator==
 - Point< T >, 11
- operator*
 - Rational< T >, 13
- Outside
 - Inside.hpp, 29
- parent
 - shortest_path.hpp, 40
 - shortest_path_time_table.hpp, 42
- PI
 - FFT.hpp, 60
- Point< T >, 10
 - length, 11
 - lengthSquared, 11
 - magnitude, 11
 - magnitudeSquared, 11
 - operator!=, 11
 - operator+, 11
 - operator-, 11
 - operator==, 11
 - x, 12
 - y, 12
- Points.hpp
 - add, 32
 - angle, 32
 - cross_product, 33
 - dist, 33
 - distSquared, 33
 - dot_product, 33
 - intersect, 33
 - length, 33
 - lengthSquared, 33
 - operator<<, 34
 - operator>>, 34

- scalar_multiplication, 34
 - simple_polygon, 34
 - subtract, 34
- Pollard_Rho
 - Factorize.hpp, 65
- Polygon_area
 - Area.hpp, 37
- Polymul
 - FFT.hpp, 61
- Polymul_Quadratic
 - FFT.hpp, 61
- powMod
 - Factorize.hpp, 65
- Prime_Numbers/Prime_Factorisation/Factorize.hpp, 63, 66
- Prime_Numbers/Prime_Sieve/Sieve.hpp, 68
- Prime_Sieve, 12
 - get_Number_Of_Primes, 12
 - get_Primes, 12
 - is_Prime, 13
 - Prime_Sieve, 12
- prime_with_mult
 - Factorize.hpp, 65
- primes
 - Factorize.hpp, 66
- Query
 - FenwickTree, 10
 - Segment_Tree< T >, 15
- query
 - DisjointSet, 7
- Rational< T >, 13
 - denominator, 14
 - normalise, 13
 - numerator, 14
 - operator+, 13
 - operator-, 14
 - operator/, 14
 - operator*, 13
- rational.hpp
 - gcd, 22
 - operator<=, 22
 - operator>=, 22
- README.md, 70
- repeatDepartures
 - Edge, 9
- reverse
 - Edge, 9
- scalar_multiplication
 - Points.hpp, 34
- search
 - TrieAutomaton, 16
- Segment_Tree
 - Segment_Tree< T >, 15
- Segment_Tree< T >, 14
 - Query, 15
 - Segment_Tree, 15
- shortest_path.hpp
 - dijkstra_shortest_path, 40
 - distance, 40
 - node, 40
 - parent, 40
 - ull, 40
 - weight, 40
- shortest_path_time_table.hpp
 - dijkstra_shortest_path_time_table, 43
 - distance, 42
 - node, 42
 - parent, 42
 - ull, 43
 - weight, 42
- simple_polygon
 - Points.hpp, 34
- Solve_System_Of_Congruencies
 - Chinese_Remainder.hpp, 17
- solver.hpp
 - equation_solver, 54
 - is_double_zero, 54
- Status
 - Inside.hpp, 29
- Strings/Aho-corasick/Aho-Corasick.hpp, 70
- Strings/Suffix-sorting/Suffix_Sorting.hpp, 72, 73
- subtract
 - Points.hpp, 34
- SuffixArray, 15
 - getSuffix, 15
 - SuffixArray, 15
- to
 - Edge, 9
- travelTime
 - Edge, 9
- TrieAutomaton, 16
 - ~TrieAutomaton, 16
 - add_string, 16
 - construct_automaton, 16
 - search, 16
 - TrieAutomaton, 16
- u128
 - Factorize.hpp, 64
 - modular.hpp, 19
- ull
 - Factorize.hpp, 64
 - modular.hpp, 19
 - shortest_path.hpp, 40
 - shortest_path_time_table.hpp, 43
- unionSets
 - DisjointSet, 7
- Update
 - FenwickTree, 10
- weight
 - Edge, 9
 - shortest_path.hpp, 40
 - shortest_path_time_table.hpp, 42

x

Point< T >, [12](#)

y

Point< T >, [12](#)