Truss structures

Submission deadline:	**2024-04-08 11:59:59**	41670.007 sec
Late submission with malus:	**2024-05-17 23:59:59** (Late submission malus: 100.0000 %)	
Evaluation:	**30.0000**	
Max. assessment:	**30.0000** (Without bonus points)	
Submissions:	9 / 120	
Advices:	0 / 0	

The task is to develop a set of classes that optimize truss structures.

Assume we need to construct a rigid structure. The structure needs to be as rigid as possible while we want to optimize its weight. One way to achieve the goal is to use a truss structure. We assume the structure is of a polygonal shape, the polygon may be non-convex. We add additional beams that connect some vertices of the polygon to enforce the polygon. To make the structure as rigid as possible, we add as many such additional beams as possible. The limitation is that the additional beams do not cross each other. In fact, the additional beams divide the polygon into disjoint triangles. The task is to find the configuration of the additional beams such that the sum of the lengths is the smallest possible. Formally, the problem is to find the minimum triangulation of the **non-convex** polygon.

The second problem is to find the total number of possible ways the triangles may be formed. For instance, a rectangle may be enforced in two ways: we add either one diagonal. The computation gets more tricky if the polygon has more vertices and has various shapes. The number of possible triangulations grows rather fast with increasing number of polygon vertices, thus the computation will be performed in custom 1024 bit wide integers.

Both problems may be solved in polynomial time with time complexity of $O(n^3)$. The complexity is rather high, thus it pays off to use more threads to speed up the computation. You are expected to develop a solution that integrates into the provided infrastructure: your code must create, schedule, synchronize, and terminate the required threads. The algorithmic solution is not needed. You may decide to implement the computation algorithm yourself or you may use a sequential solver provided in the Progtest environment.

An instance of `CPolygon` class is used to represent the problems to solve. The class contains member variables with the list of polygon vertices, the length of the minimal triangulation, and the number of triangulations. The latter two member variables are to be filled by the solver.

The problems are packed into groups, these are represented by class `CProblemPack`. The pack contains two independent list: one list contains the polygons where the triangulation needs to be computed, the second list contains the polygons where we need to compute the total number of triangulations.

Our optimizing service is viable for construction companies. We server several such companies, each such company is represented by an instance of class `CCompany`. The companies continuously generate `CProblemPack` instances to be solved. Class `CCompany` provides a method to access the next problem pack (method `waitForPack`) and a method to return the solved problem pack back (method `solvedPack`). To correctly handle the two methods, your implementation needs to start two auxiliary communication threads for each instance of `CCompany`. The first communication thread will receive the problem packs (it will call `waitForPack` in a loop and pass the received packs to further processing). The second communication thread will be used to pass the solved packs back (it will receive solved pack and call `solvedPack`). Apart from the communication, these two threads will not do any computation. The computation may take a lot of time while the communication threads are expected to promptly communicate with the company. On the other hand, the communication threads are expected to maintain the order of problem packs. The company expects to receive the solved packs in the same order it sent them. Our optimizer may (it is even expected to) read and process more than one problem pack simultaneously, however, it must preserve the order of problem packs when the solved packs are returned.

The computation is encapsulated in a `COptimizer` class instance. This class is given the references to the individual companies, it controls the execution, and it manages the worker threads. As stated above, the computation of the problems may be very time consuming. Communication threads are not intended to actually do the computation. Therefore, there will be dedicated worker threads that do the expensive computation job, leaving the communication threads free for the service of `CCompany`. A communication thread reads a `CProblemPack` instance from `CCompany::waitForPack`. The instance is passed to the worker threads, these threads compute the optimal minimal triangulation/count the possible triangulations. Once the computation is finished for all `CPolygon` instances, the completed `CProblemPack` instance is passed to the

communication thread that serves method `solvedPack` of the originating company. The communication thread is responsible for the order of the problem packs (we need to preserve the same order given by the reading) and calls the `solvedPack` method when appropriate. The number of worker threads is controlled by an external parameter, thus the computation load may be adjusted to the hardware capabilities (e.g., the number of CPUs).

The following scenario describes the expected use of class `COptimizer`:

- a new instance of `COptimizer` is created,
- the companies are created and registered (method `addCompany`),
- the computation is started (method `start`). The method is given the number of worker threads in its parameter. Method `COptimizer::start` runs the worker threads, and lets them wait for the work. Next, it runs the communication threads (two communication threads per registered company) and lets them serve the problem packs. Once all threads are initialized, the method returns to its caller,
- the communication threads receive the problem packs from the companies (`waitForPack`) and pass them to the worker threads. When `waitForPack` returns an empty smart pointer, the corresponding company is not going to provide any further problems and the communication thread may leave the loop and terminate,
- worker threads accept problems from the communication threads and solve them. When solved, the problems are passed back to the originating company (where the second communication thread returns them),
- the second communication thread receives the solved instance of `CProblemPack` and returns it to the company (method `solvedPack`). The solved packs must be returned immediately when possible (still, we have to care about the order). In particular, you cannot save the received problems in an array and return all of them in a batch at the end of the computation. The submitting communication thread terminates when the last `CProblemPack` instance is returned to the company,
- the testing environment calls `COptimizer::stop`. This call may be invoked at any moment, often even in the middle of the computation. Method `COptimizer::stop` waits until all problem packs are processed, all threads are terminated and returns to the caller,
- `COptimizer` instance is freed.

The classes and their interface:

- `CPoint` is a class that represents a vertex of a polygon. The class is very simple, there is a pair of integers representing the 2D coordinate. The class is implemented in the testing environment, you must not modify it in any way. The interface is:
  - `m_X` coordinate x,
  - `m_Y` coordinate y.
- `CPolygon` is a class that represents a single problem to optimize. It aggregates the polygon vertices, the length of the minimal triangulation, and the total number of possible triangulations. The class is implemented in the testing environment, you are not allowed to modify it in any way (technically, the testing environment implements a subclass of `CPolygon`). The interface is:
  - `m_Points` the vertices of the polygon,
  - `m_TriangMin` the length of the minimal triangulation. The length sums the length of the polygon edges and the length of the additional enforcement beams. This member variable is initialized to 0, the correct value needs to be computed and filled if the polygon is listed in the minimum triangulation problems (see below),
  - `m_TriangCnt` the total number of possible triangulations. This member variable is initialized to 0, the correct value needs to be computed and filled if the polygon is listed in the count possible triangulation problems (see below),
  - there are some auxiliary methods that simplify the handling of the vertices, see the attached code.
- `CProblemPack` is a list of problems to solve. The class is an abstract class, the testing environment implements and uses a subclass of `CProblemPack`. The implementation is fixed, you are not allowed to modify it in any way. The interface is:
  - `m_ProblemsMin` an array of problem instances; minimum triangulation needs to be computed for the listed polygons,
  - `m_ProblemsCnt` an array of problem instances; the total number of possible triangulation needs to be computed for the listed polygons,
  - methods `addMin` and `addCnt` that simplify the construction of the instance (see the attached code).
- `CCompany` is a class that represents a company. The class is an abstract class, the actual implementation is hidden in the testing environment (i.e., your program will communicate with a subclass of `CCompany`). The interface is fixed, you cannot modify it in any way. The class provides methods:
  - `waitForPack` to read the next problem pack to solve. The method returns a smart pointer encapsulating `CProblemPack` instance, or an empty smart pointer to indicate that there are no further problems to process from that particular company. The call may block for a rather long time, therefore, you must call this method from a dedicated communication thread. The communication thread is expected to call this method in a loop and it is expected to pass the received problem packs to the worker threads. The testing environment tests that this method is called by exactly one communication thread. That is, for each instance of `CCompany`,

there must exist a dedicated communication thread that calls this method,
    - `solvedPack` is a method to pass the computed instance of `CProblemPack` back to the company. The parameter is the solved instance previously read from `waitForPack`. The processing in `solvedPack` may block for a rather long time, therefore, each instance of `CCompany` must start a dedicated communication thread that receives solved problems from the worker threads and passes them back to the company. The testing environment tests that this method is called by exactly one communication thread. That is, for each instance of `CCompany`, there must exist a dedicated communication thread that calls this method. The communication thread must take care of the order the solved instances are passed to the company. The company expects the computed problem packs in the same order they were read from `waitForPack`.
- `CBigInt` is a class that implements big positive integers. The numbers are represented in a binary form, the capacity is 1024 bits. The class is implemented in the testing environment and is provided in the attached library. The implementation is limited to some basic operations:
    - set a value (either from `uint64_t`, or from a string with decimal representation),
    - convert the integer into a textual representation (a string in decimal notation),
    - addition,
    - multiplication,
    - comparison.

  There are not any other operations, in fact, no further operations are needed.
- `COptimizer` is an encapsulating class. You are expected to develop the class and implement the required interface:
    - a default constructor to initialize the instance. The constructor is not expected to start any threads,
    - method `addCompany (x)`, the method adds a new instance of the company,
    - method `start ( workThr )`, the method starts the communication and worker threads. Once the threads are started, method `start` returns to the caller,
    - method `stop`, the method waits until all problems (from all registered companies) are processed and then terminates all threads. Finally, it returns to the caller,
    - method `usingProgtestSolver()` returns `true` if you use the delivered solver (`CProgtestSolver`) or `false` if you implement the solver yourself. If you return `true`, then the testing environment does not call method `COptimizer::checkAlgorithmMin (p)` / `COptimizer::checkAlgorithmCnt (p)` below (you may leave these methods empty). Conversely, if the method returns `false`, then the testing environment disables the built-in solved in `CProgtestSolver` - the solver computes invalid results if used in this case.
    - static method `checkAlgorithmMin(p)`. The method is intended to check the correctness of the computation algorithms. A parameter to this call is an instance of `CPolygon`, the method needs to sequentially solve the minimal triangulation and save the result in the `m_TriangMin` member. This method is also used to calibrate the speed of your implementation. The testing environment measures the speed, this measurement is used to modify the size of the problems it generates. Implement this method if you do not use the built-in solver in `CProgtestSolver` (i.e., implement this method when `COptimizer::usingProgtestSolver()` returns `false`).
    - static method `checkAlgorithmCnt(p)`. The method is similar to the `checkAlgorithmMin(p)` method. The difference is this method needs to compute the number of possible triangulations and store the result into the `m_TriangCnt` member.
- `CProgtestSolver` is a class that provides a sequential solver of both triangulation problems. The class and its interface is a bit playful. This provides a good opportunity to have some extra fun when using the class. Class `CProgtestSolver` is an abstract class, the implementation is hidden in its subclass. In fact, the testing environment (and the attached library) provide two subclasses of `CProgtestSolver`: one subclass computes the minimum triangulation while the second computes the possible triangulations. The instances of the solver are created by two factory functions: `createProgtestMinSolver()` creates an instance that solves minimal triangulations and `createProgtestCntSolver()` counts the possible triangulations. `CProgtestSolver` solves the problems in batches. Each `CProgtestSolver` is assigned a certain capacity, the capacity limits the maximum number of problems the instance is willing to solve. `CProgtestSolver` instance is one-shot: you fill the problem instances and trigger the computation. When finished, the solver instance is no longer of any use (it refuses to do anything more). New `CProgtestSolver` instance must be created if there are further problems to solve. The interface is:
    - `hasFreeCapacity()` the method returns `true`, if there is a free space for next `CPolygon` instance or `false` if the solver instance is completely full,
    - `addPolygon(x)` the method adds problem `x` to solve. Return value is either `true` (problem added), or `false` (problem not added, at the capacity limit). A good strategy is to test the capacity (`hasFreeCapacity`) after each `addProblem` call. If the capacity is completely used, start the computation (method `solve()` below).
    - `solve()` the method runs the computation itself. The minimal triangulation/possible triangulations are computed and saved into the corresponding `CPolygon` instances, i.e., the computation fills in the `CPolygon::m_TriangMin` / `CPolygon::m_TriangCnt` fields. The computation does not do anything else, in particular, it does not inform the companies (does not call `CCompany::solvedPack`). Any further processing of the solved problems is left on you. Method `solve` may be called only once for `CProgtestSolver` instance, further calls end with an error. The method returns the number of solved problems, return value of 0 typically

means an error (e.g., repeated call to the `solve()` method).

`CProgtestSolver` instance has a limit on its capacity. Any attempt to exceed the capacity results in an error (`addProblem` fails). On the other hand, method `solve` may be called at any moment (however, only once for each instance). Nevertheless, do not try to abuse this and solve problem instances one by one:

- there is a limit on the created `CProgtestSolver` instances. The testing environment knows the number of problems it generates (`N`). Subsequently, it creates `CProgtestSolver` instances and sets their capacities such that the sum of the capacities is `M`. It guarantees that `M` greater or equal to `N`. However, `M` is not much bigger than `N`,
- if you used `CProgtestSolver` instances to only solve one problem, you are going to exhaust the total capacity `M` soon. There will not be any capacity left to solve the further problems,
- if you exhaust the capacity `M`, subsequent calls to `createProgtestSolver()` will return unusable `CProgtestSolver` instances (based on the Progtest mood: empty smart pointer, solver with zero capacity, or solver that fills invalid results),
- therefore, you need to fully use the capacity of the created solver instances,
- the capacities of the solvers are chosen randomly. As stated above, the solver is intended to add some fun to the programming,

Finally, the solver in the testing environment is only available in the mandatory and optional tests (it is not available in the bonus tests). If you call factory functions `createProgtestMinSolver()` / `createProgtestCntSolver()` in a bonus test, the returned smart pointer may be empty, the solver may have zero capacity, or the solver may be malfunctioning.

Submit your source code containing the implementation of class `COptimizer` with the required interface. You can add additional classes and functions, of course. Do not include function `main` nor `#include` directives to your implementation. The function `main` and `#include` directives can be included only if they are part of the conditional compile directive block (`#ifdef` / `#ifndef` / `#endif`).

Use the example implementation file included in the attached archive. Your whole implementation needs to be part of source file `solution.cpp`. If you preserve compiler directives, you can submit file `solution.cpp` as a solution.

You can use pthread or C++11 thread API for your implementation (see available #include files). The Progtest uses g++ compiler version 12.2, this version handles most of the C++11, C++14, C++17, and C++20 constructs correctly (there are some minor unsupported C++20 features).

Hints:

- Start with the threads and synchronization, use the solver from the attached library to solve the algorithmic problems. Once your program works with `CProgtestSolver`, you may replace it with your own implementation.
- To be able to use more CPU cores, serve as many problems as possible, all in parallel. You need to simultaneously receive problems from all companies, solve the problems, and submit the solved problems. Do not try to split these tasks into phases (i.e., receive all problems, then compute the problems, ...). A solution based on this principle will not work. The tests in the testing environment are designed to cause a deadlock for such solution.
- The instances of `COptimizer` are created repeatedly for various inputs. Don't rely on global variable initialization. The global variables will have different values in the second, third, and further tests. An alternative is to initialize global variables always in constructor or `start` method. Not using global variables is even better.
- Don't use mutexes and conditional variables initialized by `PTHREAD_MUTEX_INITIALIZER`. There are the same reasons as in the paragraph above. Use `pthread_mutex_init()` instead. Or use C++11 API.
- The instances of `CPolygon`, `CProblemPack`, `CCompany`, and `CProgtestSolver` are allocated by the testing environment when smart pointers are initialized. They are deallocated automatically when all references are destroyed. Don't free those instances; it is sufficient to forget all copies of the smart pointers. On the other hand, your program has to free all resources it allocates.
- Don't use `exit`, `pthread_exit` or similar calls in `stop` or in any other method. If `COptimizer::stop` method does not return back to its caller, your program will be evaluated as wrong.
- Use the sample data in the attached files. You can find an example of API calls, several test data sets, and the corresponding results there.
- The test environment uses STL. Be careful as the same STL container must not be accessed from multiple threads concurrently. You can find more information about STL parallel access in **C++ reference - thread safety.**
- The test environment has a limited amount of memory. There is no explicit limit, but the virtual machine, where tests are run has RAM size limited to 4 GiB. Your program is guaranteed at least 1 GiB of memory (i.e., data segment + stack + heap). The rest of the physical RAM is used by OS and other processes.
- If you decide to pass the bonus test, be careful to use proper granularity of parallelism. The input problem must be

divided into several subproblems to pass the bonus tests. On the other hand, if there are too many small problems, context switches induce a high overhead.
- The time intensive computation must be handled in the worker threads. The number of worker threads is determined by the parameter of method `start`. The testing environment rejects a solution that does time-intensive computation outside these threads (e.g., in the communication threads).
- There is a good reason to limit the time-intensive computation only to the worker threads. We are free to choose the number of worker threads, thus we may adjust the computer load based on the number of available CPU cores or other factors. If the time-intensive computation is done outside of the work threads (e.g., in the communication threads), the computer may be easily overloaded. We cannot easily limit the number of communication threads, the number is given by the number of companies we have to serve.

Extra hints (when developing your own solver):

- You do not need to implement your own problem solver, your implementation may use the delivered `CProgtestSolver`. The algorithms used to solve the problems are not that difficult, however, there are certain technical details that must be taken into account (dynamic programming, analytic geometry, rounding problems, floating point computation).
- `CProgtestSolver` tries to avoid floating points where possible. In particular, line segment intersections are computed in integers. This is possible since the coordinates are integers (`int`) and the computation only needs fractions with both numerator and denominator represented by an `int`. In fact the reference avoids fractions and multiplies the equations with the denominator value (yielding `long int`). Floating point numbers are only used when computing the length of the edges/additional beams.
- The problems are both convex and non-convex polygons. The algorithms that handle non-convex polygons are often more complicated and slower. The testing environment inputs only "nice" polygons where the edges do not intersect and there are not any holes inside the polygons (so called simple polygons). The algorithms are much easier thanks to this limitation.
- The minimal triangulation of a convex polygon is a popular problem that is used to demonstrate the principle of dynamic programming. The extension to non-convex polygons (simple polygons) is straightforward: the algorithms needs to add additional tests that check that division of the polygon is possible for the given pair of vertices.
- The counting of possible triangulations is again a dynamic programming application. The computation is trivial for convex polygons: the result is a Catalan number, thus there is an explicit formula. The computation for non-convex polygons is based on the idea/proof the formula is derived for convex polygons: there are some divisions of the polygon that are not allowed for non-convex polygons, thus must be eliminated.
- Your solver must be reasonably fast. The implemented solver has time complexity of $O(n^3)$ and requires $O(n^2)$ memory. The same complexities apply for both problems. However, the computation of possible triangulations is likely to have bigger multiplicative constants (it uses big numbers).
- If you correctly implement your solver and achieve the same time complexities, your solution passes all bonus tests. As stated above, the testing environment calibrates the speed of the submitted solution. The calibration is used to adjust the sizes of the generated problems. Therefore, the testing environment accepts solutions with time complexities in a certain range.
- Multithreaded solver is required to pass bonus #2 and #3. It means, the solver uses more threads to solve one instance of a polygon with many vertices.

What do the particular tests mean:

Test algoritmu (sekvencni) [Algorithm test]
The test environment calls methods `checkAlgorithmMin()` / `checkAlgorithCnt()` for various inputs and checks the computed results. The purpose of the test is to check your algorithm. No instance of `COptimizer` is created, no `start` method is called. You can check whether your implementation is fast enough with this test. The test data are randomly generated. This test is omitted if `COptimizer::usingProgtestSolver` returns `true`.

Základní test [Basic test]
The test environment creates an instance of `COptimizer` for different number of companies (C=xxx) and worker threads (W=xxx).

Základní test (W=n, C=m) [Basic test, W=n, C=m]
This test is more strict than the previous basic test. In particular, the testing environment stops the delivery of problems in the middle of the test. It waits until all pending problems are computed and returned. Once all pending problems are returned, the testing environment continues to deliver the remaining problems. If your solution does not receive/solve/submit the problems simultaneously, this test ends in a deadlock.

Test zrychleni vypoctu [Speedup test]
The test environment runs your implementation with a various number of worker threads using the same input data. The test measures the time required for the computation (wall and CPU times). As the number of worker threads increases, the wall time should decrease, and CPU time can slightly increase (the number of worker threads is below the number of physical CPU cores). If the wall time does not decrease or does not decrease enough, the test is failed. For example, the wall time shall drop to 0.5 of the sequential time if two worker threads are used. In reality,

the speedup will not be 2. Therefore, there is some tolerance in the comparison.

Busy waiting (pomale waitForPack) [Busy waiting - slow waitForPack]
: There is a sleep call inserted in between the calls to `CCompany::waitForPack` (e.g., 100 ms sleep). If the communication/worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Busy waiting (pomale solvedPack) [Busy waiting - slow solvedPack]
: There is a sleep inserted into the `CCompany::solvedPack` (e.g., 100 ms sleep). If the communication/worker threads are not synchronized/blocked properly, CPU time is increased, and the test fails.

Rozlozeni zateze #1 [Load balance #1]
: The test environment tests whether your solution uses all available worker threads to solve a single instances of `CProblemPack`. The `CProblemPack` instance contains several `CPolygon` instances, these may be solved in parallel by the existing worker threads. `CProgtestSolver` is not available in this test.

Rozlozeni zateze #2 [Load balance #2]
: The testing environment creates a single big instance of `CPolygon` and lets the optimizer compute the minimal triangulation. Your solution should use all worker threads to solve the problem, i.e., the running time shall decrease if the number of worker threads increases. If the running time does not decrease, the test is failed. `CProgtestSolver` is not available in this test.

Rozlozeni zateze #3 [Load balance #3]
: The testing environment creates a single big instance of `CPolygon` and lets the optimizer compute the possible triangulations. Your solution should use all worker threads to solve the problem, i.e., the running time shall decrease if the number of worker threads increases. If the running time does not decrease, the test is failed. `CProgtestSolver` is not available in this test.

Update 2024-03-09:

- the correct method name is `CProgtestSolver::addPolygon` (the incorrect wording was `CProgtestSolver::addProblem`).

Update 2024-03-14:

- the attached archive contains progtest solver library compiled for aarch64-linux.

Sample data:		**Download**
Submit:	Browse... **No file selected.**	**Submit**

Reference

- **Evaluator: computer**
  - Program compiled
  - Test 'Test algoritmu (triangMin, sekvencni)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.007 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Test algoritmu (triangCnt, sekvencni)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.057 s (limit: 3.993 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Zakladni test (W=1, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.397 s (limit: 40.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Zakladni test (W=n, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.246 s (limit: 39.603 s)
    - CPU time: 0.922 s (limit: 39.587 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Zakladni test (W=1, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.379 s (limit: 39.357 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Zakladni test (W=n, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.247 s (limit: 38.978 s)

        - CPU time: 0.994 s (limit: 38.262 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Test zrychleni vypoctu (triangMin)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 3.068 s (limit: 38.731 s)
        - CPU time: 6.721 s (limit: 37.268 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Test zrychleni vypoctu (triangCnt)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 4.304 s (limit: 35.663 s)
        - CPU time: 9.416 s (limit: 30.547 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Test zrychleni vypoctu (triangMin+triangCnt)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 3.227 s (limit: 31.359 s)
        - CPU time: 7.175 s (limit: 21.131 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Busy waiting test (pomale waitForPack)': success
        - result: 100.00 %, required: 50.00 %
        - Total run time: 2.713 s (limit: 15.000 s)
        - Optional test success, evaluation: 100.00 %
    - Test 'Busy waiting test (pomale solvedPack)': success
        - result: 100.00 %, required: 50.00 %
        - Total run time: 2.560 s (limit: 12.287 s)
        - Optional test success, evaluation: 100.00 %
    - Test 'Busy waiting test (pomale waitForPack i solvedPack)': success
        - result: 100.00 %, required: 50.00 %
        - Total run time: 2.862 s (limit: 9.727 s)
        - Optional test success, evaluation: 100.00 %
    - Test 'Rozlozeni zateze #1': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.852 s (limit: 10.000 s)
        - CPU time: 3.393 s (limit: 10.000 s)
        - Bonus test - success, evaluation: 120.00 %
    - Test 'Rozlozeni zateze #2': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.368 s (limit: 10.000 s)
        - CPU time: 0.652 s (limit: 10.000 s)
        - Bonus test - success, evaluation: 120.00 %
    - Test 'Rozlozeni zateze #3': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.366 s (limit: 9.632 s)
        - CPU time: 0.646 s (limit: 9.348 s)
        - Bonus test - success, evaluation: 120.00 %
    - Overall ratio: 172.80 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.20 * 1.20 * 1.20)
- Total percent: 172.80 %
- Early submission bonus: 3.00
- Total points: 1.73 * (30.00 + 3.00) = 57.02

		Total	Average	Maximum	Function name
SW metrics:	Functions:	**47**	--	--	--
	Lines of code:	**529**	**11.26 ± 10.94**	**47**	`CProblemJob::lineLineInt`
	Cyclomatic complexity:	**164**	**3.49 ± 5.35**	**24**	`CProblemCntJob::fillTile`

9	**2024-04-08 02:10:48**	**Download**
Submission status:	Evaluated	
Evaluation:	30.0000	

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.005 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.034 s (limit: 3.995 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.322 s (limit: 40.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=n, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.200 s (limit: 39.678 s)
    - CPU time: 0.683 s (limit: 39.677 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.507 s (limit: 39.478 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=n, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.215 s (limit: 38.971 s)
    - CPU time: 0.849 s (limit: 38.474 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Speedup test (triangMin)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 2.895 s (limit: 38.756 s)
    - CPU time: 5.598 s (limit: 37.625 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Speedup test (triangCnt)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 3.682 s (limit: 35.861 s)
    - CPU time: 6.954 s (limit: 32.027 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Speedup test (triangMin+triangCnt)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 3.096 s (limit: 32.179 s)
    - CPU time: 6.047 s (limit: 25.073 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Busy waiting test (slow waitForPack)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 2.875 s (limit: 15.000 s)
    - Optional test success, evaluation: 100.00 %
  - Test 'Busy waiting test (slow solvedPack)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 2.721 s (limit: 12.125 s)
    - Optional test success, evaluation: 100.00 %
  - Test 'Busy waiting test (slow both waitForPack and solvedPack)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 3.635 s (limit: 9.404 s)
    - Optional test success, evaluation: 100.00 %
  - Test 'Load balance #1': failed
    - result: 0.00 %, required: 100.00 %
    - Total run time: 0.006 s (limit: 10.000 s)
    - Bonus test - failed, evaluation: No bonus awarded
    - Failed (invalid output)
    - Failed (invalid output)
    - Failed (invalid output)

        - Failed (invalid output)
    - Test 'Load balance #2': failed
        - result: 0.00 %, required: 100.00 %
        - Total run time: 0.003 s (limit: 10.000 s)
        - Bonus test - failed, evaluation: No bonus awarded
        - Failed (invalid output)
        - Failed (invalid output)
        - Failed (invalid output)
    - Test 'Load balance #3': failed
        - result: 0.00 %, required: 100.00 %
        - Total run time: 0.001 s (limit: 9.997 s)
        - Bonus test - failed, evaluation: No bonus awarded
        - Failed (invalid output)
        - Failed (invalid output)
        - Failed (invalid output)
    - Overall ratio: 100.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00)
- Total percent: 100.00 %
- Total points: 1.00 * 30.00 = 30.00

SW metrics:		Total	Average	Maximum	Function name
	Functions:	**21**	--	--	--
	Lines of code:	**224**	**10.67 ± 13.00**	**62**	`COptimizer::receiver`
	Cyclomatic complexity:	**55**	**2.62 ± 2.55**	**12**	`COptimizer::receiver`

8	**2024-04-08 01:00:50**	**Download**
Submission status:	Evaluated	
Evaluation:	0.0000	

- **Evaluator: computer**
    - Program compiled
    - Test 'Algorithm test (triangMin, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.004 s (limit: 4.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Algorithm test (triangCnt, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.030 s (limit: 3.996 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.460 s (limit: 40.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.500 s (limit: 39.540 s)
        - CPU time: 0.795 s (limit: 39.539 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=m)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.528 s (limit: 39.040 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=m)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.647 s (limit: 38.512 s)
        - CPU time: 1.080 s (limit: 38.200 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Speedup test (triangMin)': failed
        - result: 33.33 %, required: 100.00 %
        - Total run time: 10.569 s (limit: 37.865 s)

        - Mandatory test failed, evaluation: 0.00 %
        - Failed (invalid output)
        - Failed (invalid output)
    - Overall ratio: 0.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**248**	**11.81 ± 14.64**	62	`COptimizer::receiver`
Cyclomatic complexity:	**61**	**2.90 ± 2.89**	12	`COptimizer::receiver`

7	**2024-04-08 00:41:31**	**Download**
Submission status:	Evaluated	
Evaluation:	0.0000	

- **Evaluator: computer**
    - Program compiled
    - Test 'Algorithm test (triangMin, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.005 s (limit: 4.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Algorithm test (triangCnt, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.030 s (limit: 3.995 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.388 s (limit: 40.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.597 s (limit: 39.612 s)
        - CPU time: 0.914 s (limit: 39.611 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=m)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.527 s (limit: 39.015 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=m)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.709 s (limit: 38.488 s)
        - CPU time: 1.209 s (limit: 38.155 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Speedup test (triangMin)': failed
        - result: 33.33 %, required: 100.00 %
        - Total run time: 9.390 s (limit: 37.779 s)
        - Mandatory test failed, evaluation: 0.00 %
        - Failed (invalid output)
        - Failed (invalid output)
    - Overall ratio: 0.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**248**	**11.81 ± 14.64**	62	`COptimizer::receiver`
Cyclomatic complexity:	**61**	**2.90 ± 2.89**	12	`COptimizer::receiver`

6	2024-04-08 00:37:55	Download
Submission status:	Evaluated	
Evaluation:	0.0000	

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.004 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.030 s (limit: 3.996 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.450 s (limit: 40.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=n, C=1)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.471 s (limit: 39.550 s)
    - CPU time: 0.860 s (limit: 39.549 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.539 s (limit: 39.079 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=n, C=m)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.686 s (limit: 38.540 s)
    - CPU time: 1.145 s (limit: 38.137 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Speedup test (triangMin)': failed
    - result: 33.33 %, required: 100.00 %
    - Total run time: 10.084 s (limit: 37.854 s)
    - Mandatory test failed, evaluation: 0.00 %
    - Failed (invalid output)
    - Failed (invalid output)
  - Overall ratio: 0.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**245**	**11.67 ± 14.16**	**59**	`COptimizer::receiver`
Cyclomatic complexity:	**61**	**2.90 ± 2.89**	**12**	`COptimizer::receiver`

5	2024-04-07 23:11:42	Download
Submission status:	Evaluated	
Evaluation:	0.0000	

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.005 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success

        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.034 s (limit: 3.995 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.435 s (limit: 40.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.647 s (limit: 39.565 s)
        - CPU time: 0.966 s (limit: 39.564 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=m)': Abnormal program termination (Time limit exceeded)
        - Cumulative test time exceeded, killed after:: 38.965 s (limit: 38.918 s)
        - Mandatory test failed, evaluation: 0.00 %
    - Overall ratio: 0.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**210**	**10.00 ± 12.47**	**60**	`COptimizer::receiver`
Cyclomatic complexity:	**54**	**2.57 ± 2.56**	**12**	`COptimizer::receiver`

4 **2024-04-07 22:39:25** **Download**

Submission status: Evaluated
Evaluation: 0.0000

- **Evaluator: computer**
    - Program compiled
    - Test 'Algorithm test (triangMin, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.005 s (limit: 4.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Algorithm test (triangCnt, sequential)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.034 s (limit: 3.995 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=1, C=1)': success
        - result: 100.00 %, required: 100.00 %
        - Total run time: 0.436 s (limit: 40.000 s)
        - Mandatory test success, evaluation: 100.00 %
    - Test 'Basic test (W=n, C=1)': failed
        - result: 50.00 %, required: 100.00 %
        - Total run time: 0.624 s (limit: 39.564 s)
        - CPU time: 0.931 s (limit: 39.563 s)
        - Mandatory test failed, evaluation: 0.00 %
        - Failed (invalid output)
    - Overall ratio: 0.00 % (= 1.00 * 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**210**	**10.00 ± 12.47**	**60**	`COptimizer::receiver`
Cyclomatic complexity:	**52**	**2.48 ± 2.54**	**12**	`COptimizer::receiver`

3 **2024-04-07 22:08:21** **Download**

Submission status:	Evaluated
Evaluation:	0.0000

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.005 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.030 s (limit: 3.995 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=1)': failed
    - result: 50.00 %, required: 100.00 %
    - Total run time: 0.379 s (limit: 40.000 s)
    - Mandatory test failed, evaluation: 0.00 %
    - Failed (invalid output)
  - Overall ratio: 0.00 % (= 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**210**	**10.00 ± 12.47**	**60**	`COptimizer::receiver`
Cyclomatic complexity:	**52**	**2.48 ± 2.54**	**12**	`COptimizer::receiver`

2	**2024-04-07 21:54:26**	**Download**
Submission status:	Evaluated	
Evaluation:	0.0000	

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.004 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.035 s (limit: 3.996 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=1)': failed
    - result: 50.00 %, required: 100.00 %
    - Total run time: 0.495 s (limit: 40.000 s)
    - Mandatory test failed, evaluation: 0.00 %
    - Failed (invalid output)
  - Overall ratio: 0.00 % (= 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**21**	--	--	--
Lines of code:	**210**	**10.00 ± 12.47**	**60**	`COptimizer::receiver`
Cyclomatic complexity:	**52**	**2.48 ± 2.54**	**12**	`COptimizer::receiver`

1	**2024-04-04 22:59:07**	**Download**

Submission status:	Evaluated
Evaluation:	0.0000

- **Evaluator: computer**
  - Program compiled
  - Test 'Algorithm test (triangMin, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.005 s (limit: 4.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Algorithm test (triangCnt, sequential)': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.030 s (limit: 3.995 s)
    - Mandatory test success, evaluation: 100.00 %
  - Test 'Basic test (W=1, C=1)': Abnormal program termination (Segmentation fault/Bus error/Memory limit exceeded/Stack limit exceeded)
    - Total run time: 1.366 s (limit: 40.000 s)
    - Mandatory test failed, evaluation: 0.00 %
  - Overall ratio: 0.00 % (= 1.00 * 1.00 * 0.00)
- Total percent: 0.00 %
- Total points: 0.00 * 30.00 = 0.00

SW metrics:

	Total	Average	Maximum	Function name
Functions:	**15**	--	--	--
Lines of code:	**159**	**10.60 ± 8.28**	**33**	`COptimizer::receiver`
Cyclomatic complexity:	**43**	**2.87 ± 2.22**	**8**	`COptimizer::receiver`