Rental service

Submission deadline:	**2023-04-17 11:59:59**	2644732.423 sec
Evaluation:	**0.0000**	
Max. assessment:	**30.0000** (Without bonus points)	
Submissions:	0 / 75	
Advices:	0 / 0	

The task is to develop a set of classes that optimize the profit of a rental company.

Assume a rental company that rents tools. The company owns some tools to rent, it may own more than one item of a certain tool. For instance, the company may own 1 excavator, 3 vibratory plates, 2 pneumatic drills, ... The rental company uses online reservation systems where the customers place their requests. The company, however, uses innovative business models - it rents the tools in an auction inspired system. Each customer places his request with the time frame (interval from-to) and the offered payment. The rental company then chooses some requests such that it maximizes its profit. The choice follows the following limitations:

- if we own `n` items of a tool, we may serve at most `n` customers simultaneously,
- the intervals chosen for each item must not overlap/touch,
- the rental company is not very rich, it owns at most 3 items of a certain tool.

Although the optimal choice of intervals may seem trivial, it is indeed a rather complex problem. The solution may be found using dynamic programming, the algorithm has been presented in BI-AG1. Specifically, the algorithm is not a greedy algorithm. Therefore, any algorithm that chooses the intervals based on some greedy criterion will fail (e.g., choose the non-overlapping intervals starting from the highest payment or from the best payment-to-duration ratio). The time complexity heavily depends on the number of items we own and rent, the time complexities are $O(n \log(n))$, $O(n^2)$, and $O(n^3)$ for 1, 2, and 3 items, respectively. Since the problems are quite complex, we will use threads to speed up the computation.

Consider the following input requests to rent a certain tool:

```
from  to   payment
 0 -  6    8
 7 -  9    2
10 - 13    2
 0 -  1    4
 2 - 11    4
12 - 13    4
 0 -  3    2
 4 -  5    2
 8 - 13    8

The optimal schedule if we rent just 1 item:
1:  0-6 8-13
profit: 16

The optimal schedule if we rent 2 items:
1:  0-6 7-9 12-13
2:  0-1 4-5 8-13
profit: 28

The optimal schedule if we rent 3 items:
1: 0-6 7-9 10-13
2: 0-1 2-11 12-13
3: 0-3 4-5 8-13
profit: 36
```

Your solution is expected to integrate into the computation infrastructure (see below) and it must correctly handle threads (create, schedule, synchronize, terminate). 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.

Each rental problem to solve is represented by an instance of `CProblem` class. The class contains the requests from our customers and the number of items we own (rent).

The problems are packed into groups, these are represented by class `CProblemPack`. For instance, the company may rent various tools (excavators, drills, ...), thus there will be `CProblem` instance for each kind of tool the company rents.

Our optimizing service may serve more than one rental company. Each rental company is represented by an instance of class `CCompany`. The rental 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 rental companies, it controls the execution, and it manages the worker threads. As stated above, the computation of the schedule 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 rental schedule and compute the maximum profit. Once the computation is finished for all `CProblem` instances, the completed `CProblemPack` instance is passed to the communication thread that serves method `solvedPack` of the originating rental 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 (# of CPUs).

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

- a new instance of `COptimizer` is created,
- the rental 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 rental 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 rental companies (`waitForPack`) and pass them to the worker threads. When `waitForPack` returns an empty smart pointer, the corresponding rental 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 rental company (where the second communication thread returns them),
- the second communication thread receives the solved instance of `CProblemPack` and returns it to the rental 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 rental company,
- the testing environment calls `COptmizer::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,
- `COptmizer` instance is freed.

The classes and interfaces:

- `CInterval` is a simple class that encapsulates a single request from a customer. The class is implemented in the testing environment, you must not modify it in any way. The interface is:
  - `m_From` the beginning of the rent interval,
  - `m_To` the end of the rent interval,
  - `m_Payment` the offered payment.

  You may rely on `m_From` < `m_To`.
- `CProblem` is a class that represents a single problem to optimize. It aggregates the intervals (requests from the customers), and the number of tools items we rent. 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 `CProblem`). The interface is:

    - `m_Intervals` the requests from the customers,
    - `m_Count` the number of tool items we have (and rent),
    - `m_MaxProfit` the maximal profit we may achieve (filled by the computation),
    - there are some auxiliary methods that simplify the handling of the lists/maps, 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_Problems` an array of problem instances to solve,
    - method `add (problem)` simplifies the construction of the instance (see the attached code).
- `CCompany` is a class that represents a rental 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 rental 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 rental 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 rental 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 rental company. The rental company expects the computed problem packs in the same order they were read from `waitForPack`.
- `COptmizer` 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 rental 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 rental 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::checkAlgorithm(problem)` below (you may leave the method 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 `checkAlgorithm(problem)`. The method is intended to check the correctness of the computation algorithms. A parameter to this call is an instance of `CProblem` to solve. The method sequentially solves the problems and fills the result in the `CProblem` instance. 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`).
- `CProgtestSolver` is a class that provides a sequential solver of the scheduling 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. `CProgtestSolver` instances are created by a factory function `createProgtestSolver()`. `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 instance is no longer of any use (it refuses to 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 `CProblem` instance or `false` if the solver instance is completely full,
    - `addProblem(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 optimal schedule is computed for all stored `CProblem` instances, i.e., the computation fills in the `CProblem::m_MaxProfit` fields. The computation does not do anything else, in particular, it does not inform the rental 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 0 typically means an error

(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 function `createProgtestSolver()` 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 task solution.

You can use pthread or C++11 thread API for your implementation (see available #include files). The Progtest uses g++ compiler version 10.2, this version handles most of the C++11, C++14, and C++17 constructs correctly. The compiler has a limited support for C++20, however, we do not recommend to use C++20.

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 `CProblem`, `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 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 reference solution limits the maximum number of problems concurrently solved by the worker threads to mitigate this 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):

- The solver is not trivial. The algorithm must use some form of dynamic programming and there are some technical details that must be taken into account. Of them, the most important are the memory requirements of your algorithm.
- Your solver must be reasonably fast and must use reasonable amount of memory. Both running time and memory requirements depend on the number of tools we may rent. The solution implemented in `CProgtestSolver` has the following requirements (`n` is the number of intervals):

m_Count	T(n)	M(n)
1	$O(n^2)$	$O(n)$
2	$O(n^3)$	$O(n^2)$
3	$O(n^4)$	$O(n^3)$

- The solver provided in `CProgtestSolver` class is not time-optimal. If you correctly implement your solver and achieve the same time complexities, your solution passes all bonus tests. Of course, the time-optimal solution passes too (such solution would require more tricks in the algorithmic design however). 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 solution for problems with `m_Count=2` is required to pass bonus #2. Similarly, multithreaded solution for problems with `m_Count=3` is required to pass bonus #3. There is no need to use more threads to solve problems with `m_Count=1`, the generated problems are too small.
- The testing environment adjusts `n` with respect to the memory and time requirements. Bonus test #2 (`m_Count=2`) sets `n` to approx. 1000, bonus test #3 (`m_Count=3`) sets `n` to about 200.
- A good starting point is a sequential algorithm that solves the problem instance with time/memory requirements from the above table. Such solution is sufficient to pass bonus #1.
- The sequential solution must be converted into multithreaded to pass bonus tests #2 and #3. An intermediate step may be an algorithm with higher memory requirements ($O(n^3)$ for `m_Count=2` and $O(n^4)$ for `m_Count=3`). Such solution is not going to pass the tests (there is not enough RAM in the testing environment), moreover, the time efficiency may drop (allocation overhead, cache efficiency, ...).
- The algorithms from the previous step may be further optimized and the memory requirements may be decreased by a factor of `n`. Analysis of data dependencies is the key; some data are not needed for the entire duration of the computation.

What do the particular tests mean:

Test algoritmu (sekvencni) [Algorithm test]
: The test environment calls methods `checkAlgorithm()` 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 `CProblem` 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 instance of `CProblem` with `m_Count=2`. 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 instance of `CProblem` with `m_Count=2`. 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 Mar 12: new version of progtest_solver library. Compiled and tested on:

- arm64-darwin22: Homebrew GCC 12.2.0, macOS Venture 13.2.1,
- x86_64-darwin22: Homebrew GCC 12.2.0, macOS Venture 13.2.1,
- aarch64-linux-gnu: Linux Ubuntu 22.04.2 LTS (Jammy Jellyfish aarch64)

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

☐ **Reference**