Olympiads in Informatics, 2016, Vol. 10, 177–194
© 2016 *IOI, Vilnius University*
DOI: 10.15388/ioi.2016.11

Distributed Tasks: Introducing Distributed Computing to Programming Competitions

Adam KARCZMARZ[1], Jakub ŁĄCKI[2], Adam POLAK[3]
Jakub RADOSZEWSKI[1,4], Jakub O. WOJTASZCZYK[5]
[1]*Faculty of Mathematics, Informatics and Mechanics, University of Warsaw*
ul. Banacha 2, 02-097 Warsaw, Poland
[2]*Department of Computer, Control, and Management Engineering Antonio Ruberti at Sapienza University of Rome*
via Ariosto 25, 00185 Roma, Italy
[3]*Department of Theoretical Computer Science, Faculty of Mathematics and Computer Science, Jagiellonian University*
ul. Łojasiewicza 6, 30-348 Kraków, Poland
[4]*King's College London*
Strand, London WC2R 2LS
[5]*Google Warsaw*
Emilii Plater 53, 00-113 Warsaw, Poland
e-mail: a.karczmarz@mimuw.edu.pl, j.lacki@mimuw.edu.pl, polak@tcs.uj.edu.pl, jrad@mimuw.edu.pl, onufry@google.com

Abstract. In this paper we present *distributed tasks*, a new task type that can be used at programming competitions. In such tasks, a contestant is supposed to write a program which is then simultaneously executed on multiple computing nodes (machines). The instances of the program may communicate and use the joint computing power to solve the task presented to the contestant. We show a framework for running a contest with distributed tasks, that we believe to be accessible to contestants with no previous experience in distributed computing. Moreover, we give examples of distributed tasks that have been used in the last two editions of a Polish programming contest, Algorithmic Engagements, together with their intended solutions. Finally, we discuss the challenges of grading and preparing distributed tasks.

Keywords: programming contests, distributed tasks.

1. Introduction

When looking at major programming competitions, it is easy to notice that a large number of them are very similar in design. To name a few, the IOI, the ACM ICPC, Google Code Jam, TopCoder's algorithmic track, Facebook Hacker Cup, the CodeForces competitions and many more, are all focused on small, self-contained tasks, with automated judging based on testing a program on a judge-provided set of testcases and mostly algorithmic in nature.

This model seems attractive both to the organizers and the participants. The organizers appreciate the very clear, non-subjective mechanism of judging and the automation of judging, which means the competition can scale out easily to a larger number of competitors. The participants also appreciate the automated judging (which means fast results) and the objective judgment criteria (which makes the competition more fair); additionally the entry barrier to such contests is pretty low, as the introductory-level tasks can be very simple.

If we consider programming competitions a way of educating future computer scientists and professional programmers, the model has its strengths, but also weaknesses. Some of these weaknesses come from the very nature of the small, self-contained tasks (which are a large part of the model's attractiveness): the participants do not learn about code maintainability and extensibility, they also do not learn anything about larger-scale system design. These weaknesses seem to be intrinsic to the model itself; and competitions that abandon the model moving to larger or less clear-cut tasks are frequently less successful (for example, TopCoder's Marathon Match track has over 10 times less registered participants than the Algorithm track).

However, there are also areas of programming expertise that the existing competitions do not teach, which are less intrinsically tied to the model of these competitions. In particular, the focus on algorithmic problems is not crucial to the model of the competition itself. Indeed, multiple authors already explored extending this classical model of competitions to other fields of computer science, for example visualization and precomputation (Kulczyński *et al.*, 2011), online algorithms (Komm, 2011), computer graphics and cryptography (Forišek, 2013).

1.1. *Distributed Programming*

Distributed (and cloud) computing has gained importance quickly in the recent years. The computing giants of today – Google, Amazon, Facebook and others – do not operate on the enormous mainframes that dominated computing in the past, but on networked farms of smaller servers. This is a model of computing that is not inherently in conflict with the algorithmic programming contest model, but is not taught by the dominant competitions of today; students that gained most of their programming skills through programming competitions will be totally unfamiliar with even the basic paradigms of distributed computing like the MapReduce framework (Dean and Ghemawat, 2008) or the CAP theorem (Fox and Brewer, 1999).

In this article we present a framework for running a contest focusing on distributed algorithms, developed by engineers in Google's Warsaw office in collaboration with the University of Warsaw, and show sample problems used in the Algorithmic Engagements (*Potyczki Algorytmiczne* in Polish) contest ran by the University of Warsaw. The same framework is used at the recently introduced Google's Distributed Code Jam competition.

1.2. *Designing a Distributed Programming Contest*

The primary focus of our design was simplicity from the contestant's point of view. The introduction of a new programming paradigm, likely unfamiliar to most participants, is clearly a challenge for the contestants, and we aimed at making the transition as smooth as possible.

Thus, the basic interface is similar to a programming contest like IOI. The participant submits a single program, which is compiled and executed by the framework. The same program is run on every *node* (computer) available to the participant.

Obviously, the nodes need to be able to communicate in order to collaborate in computing. We decided on a protocol based on simple message-passing ("send this array of bytes to this node"). The message passing methods are available to the program through a library that is common for all problems, and provided by the framework.

We also considered using an RPC-style interface. This, however, is more complex on an API level. A standard approach to RPC is that the programmer has to declare an interface (which will contain the remotely-callable methods). The server will just implement the methods of this interface. On the client side, the infrastructure needs to provide a way to generate a "stub" connected to some particular server; this stub will automagically have the method calls autogenerated. The language would have to provide some way to annotate that the interface is to be treated as a "remote" interface, increasing the complexity of the infrastructure implementation, and meaning more "magic" happening under the hood, which – in our perception – decreases the comprehensibility of the system. For instance, the "stub implementation" would need to make a choice of deep versus shallow copying of the arguments, each choice potentially leading to confusion for the participants.

Let us now present the functions of our library, together with their declarations in C++.

First, the library provides a function that returns the number of nodes M on which the solution is running, and the index (in the range $[0, M - 1]$) of the node on which the calling process is running.

```
int NumberOfNodes();
int MyNodeId();
```

The library maintains in each node a message buffer for each of the M nodes, which represents messages that are to be sent to this node. Messages are added to the buffer through the `Put`-methods.

```
// Append "value" to the message that is being prepared for
// the node with id "target". The "Int" in PutInt is
// interpreted as 32 bits, regardless of whether the actual
// int type will be 32 or 64 bits.
void PutChar(int target, char value);
void PutInt(int target, int value);
void PutLL(int target, long long value);
```

There is also a method that sends the message that was accumulated in the buffer for a given node and clears this buffer. This method is non-blocking, that is, it does not wait for the receiver to call `Receive`, but returns immediately after sending the message.

```
void Send(int target);
```

The following function is used for receiving messages.

```
int Receive(int source);
```

The library has a receiving buffer for each remote node. When you call `Receive` and retrieve a message from a remote node, the buffer tied to this remote node is overwritten. You can then retrieve individual parts of the message through the `Get`-methods. This method is blocking – if there is no message to receive, it will wait for the message to arrive.

You can call `Receive(-1)` to retrieve a message from any source, or set `source` to a number from $[0,\ M-1]$ to retrieve a message from a particular source. `Receive` returns the number of the node which sent the message, which is equal to `source`, unless `source` is `-1`.

Finally, for reading the buffer of incoming messages, the following three methods are provided.

```
// The "Int" in GetInt is interpreted as 32 bits, regardless
// of whether the actual int type is 32 or 64 bits.
char GetChar(int source);
int GetInt(int source);
long long GetLL(int source);
```

Each of these methods returns and consumes one item from the buffer of the appropriate node. You must call these methods in the order in which the elements were appended to the message (so, for instance, if the message was created with `PutChar`, `PutChar`, `PutLL`, you must call `GetChar`, `GetChar`, `GetLL` in this order). If you call them in a different order, or you call a `Get`-method after consuming all the contents of the buffer, the behaviour is undefined.

The serialization we decided to use is very basic, compared to models like Java's serialization mechanisms, Python's "pickle" or even Google's protocol buffer language. Again, we preferred to err on the side of simplicity, to minimize the entry barrier – this simple language turns out to be easy enough for the simple concurrency required to solve our problems, and is more straightforward to understand, both in terms of "how big will be the serialized message", and "what is actually serialized" (this, again, is the deep vs shallow copy question).

For correctness of execution, we chose what seemed the most natural model. The backend guarantees failure-less execution on all nodes, and requires all the instances of the program to execute correctly, within the specified time and memory limits. Note that the time used by the program is measured from the moment when the instances start until all instances have finished execution. This assumption is justified with the following example involving two machines. The first one spends second of CPU time and then sends a message to the second one. The second machine first waits for a message from the first machine and then uses one second of CPU time. The total time used by the solution is then roughly two seconds, although each instance used only one second of CPU time.

In most of programming competitions, the programs access the input data by reading from standard input. (TopCoder's Algorithm contest breaks out of this by providing the input as an argument to the function the contestant is supposed to write.) However, a large fraction of the interesting distributed problems admit a solution that runs in $O(N/M)$ time, where N is the size of the input, and M is the number of nodes available to the contestant. This implies, in particular, that no node can afford to read the whole input (since that alone would take $O(N)$ time). Standard input is only accessible in a linear fashion, so it is not feasible for providing very large input data.

The approach we chose is what is typically done for interactive tasks on competitions like IOI (see, e.g., Chávez, 2015) – each problem with large inputs defines a set of input-access methods that are available to the program, similarly to the message-passing interface. These methods are guaranteed to return the same values on all nodes (so, each node has access to the same view of the input). Additionally, we provide upper bounds on the execution time of a single call to the input-providing methods.

Output is much simpler to handle (as it is often much smaller), so we went with the standard programming contest practice of expecting the output to be provided via standard output. We expect exactly one node to produce the output, while the others should not produce anything. This is a somewhat arbitrary decision (we could equally well have all the nodes output the exact same data to the standard output), however, as many solutions in practice have some sort of a "master" node that aggregates the work of the other nodes, it is convenient to the contestant to have one node output the result of the computation, and for the infrastructure not to prescribe which of the nodes it is.

As for the number of nodes we run the contestants' solutions on, we chose $M = 100$. This is large enough that a speedup by a factor of M is big enough to offset the extra time needed for inter-node communication, and yet small enough that providing that many nodes for judging is actually feasible.

The final issue that needs to be considered is the amount of data sent during the communication between the nodes, both in terms of the number of individual messages sent, and the size of those messages. Obviously, the limits here will be dependent on the infrastructure we run the contest on. Benchmarks on our framework show that a single message will take roughly 3–5ms (split between the processing time in the sender, the actual network latency and the processing time on the receiver). This number will be constant for messages from negligible to tens of kilobytes, and start growing linearly when the message size goes into hundreds of kilobytes.

While in the end, to fine-tune a solution the contestant will need to understand these patterns (for this purpose we provided the results of a few benchmarks to the contestants), we wanted the basic usecase not to require dealing with the calculations. To this end, we introduced an upper limit on the total number (around 1000) and size (a few megabytes) of messages a single node can send within the time limit, which roughly corresponds to the total throughput it could achieve if it spent all its time on communication. This is a clear and concise way of informing the participants that if their solution stays well within those bounds, its performance should not be significantly affected by the communication overhead.

1.3. *Distributed Contest Judging Infrastructure*

From a research perspective, the most interesting challenge in preparing such a distributed programming contest is defining the exact programming model from the contestants' point of view. However, when doing it in practice, there is also a considerable amount of engineering effort involved in providing the infrastructure to run such a contest.

In the process of preparing problems the most interesting new challenge is writing a library that provides the input data. While in a standard programming contest the input is just a text file, and can be generated offline in an arbitrary fashion and almost arbitrary time, in the distributed contests the requirements are much more strict. The input-providing library needs to satisfy the following requirements:

- Access to an arbitrary element.
- Consistency across nodes, and across accesses.
- Access times on the order of 100ns.
- Ability to serve data with total size on the order of 10GB or more.

The requirements stem from the input model we chose. In particular, if we want to allow $O(N/M)$-time solutions, with runtimes on the order of 1–5 seconds, we need the input read by one node to be on the order of at least 10^7 items (which means 10^9 items in total – if each item is, say, two 64-bit integers, we get a total of 16GB), and we need the node to be able to read these 10^7 items within 1 second (so that the input reading does not dominate the computation).

The access times coupled with the data size mean it is infeasible to pregenerate all the input data – 10GB is too much to conveniently store in memory, while disk access (and even SSD access) is too slow. Thus, the input data is generated on the fly. This requires using pseudo-random generators that generate a sequence of numbers, and provide consistent and fast access to each element of the sequence (e.g., the CityHash family of functions).

In the judging system, the challenge is the scale. Judging a single testcase for a single solution requires, typically, 100 virtual machines. So, as an example, a deployment of 900 virtual machines on Google Compute Engine were used as the backend for the Online Round of Google's Distributed Code Jam. We think it is interesting that cloud computing, which is making distributed computing important as a topic of programming competitions, is also making distributed programming competitions much easier to organize: instead of buying physical hardware to support such a competition, it is easy to rent virtual machines and pay by the minute.

However, the real challenge in setting up a new competition type is in finding attractive problems: ones that challenge the contestants' creativity and problem-solving skill, without requiring significant domain knowledge (in this case – knowledge of distributed programming paradigms), and that are fun, but not tedious, to implement, once you have the correct set of ideas. In the rest of this paper, we provide examples of problems that – in our opinion – satisfy these requirements.

We start with a simple task which lets us demonstrate the basics of the framework from the contestant's point of view. The remaining tasks were presented during the Algorithmic Engagements contest; their authors are: Jakub Łącki, Jakub Wojtaszczyk (task "Workshop"); Jakub Łącki (task "Assistant"); Adam Karczmarz (task "Sabotage").

2. Sample Task "Divisors"

In this task we are to count the number of divisors of a given positive integer $n \leq 10^{18}$.

Input

In this task the input data is provided via the standard input. The only line of the standard input contains n.

Output

Your program should print exactly one line to the standard output containing one integer: the number of divisors of n.

2.1. *Solution*

The easiest (sequential) solution that we can come up with is to check all candidates for a divisor d up to $\sqrt{n}$ and their counterparts of the form $\frac{n}{d}$. A sample C++ code follows.

```
int main() {
  long long n;
  int divisors_num = 0;
  cin >> n;
  for (long long d = 1; d * d <= n; ++d) {
    if (n % d == 0) {
      ++divisors_num;
      if (n / d != d)
        ++divisors_num;
    }
  }
  cout << divisors_num << endl;
}
```

Probably this is not the fastest sequential solution for this problem. We will, however, focus on how to speed it up by performing the computations using M nodes (machines).

A natural idea is to partition the set of possible divisors and let each of the machines look through each of the parts.

One way of performing this partition is as follows: machine 0 gets to check the candidates 1, $1 + M$, $1 + 2M$, $\dots$, machine 1 gets to check the candidates 2, $2 + M$, $2 + 2M$, $\dots$, etc. To make it work, it suffices to change the for-loop from the above code to:

```
for (long long d = 1 + MyNodeId(); d * d <= n; d +=
  NumberOfNodes()) {
```

In the end we need to aggregate the partial results. Let us select any of the machines (say, machine 0) as an aggregator. We just need to add code to be executed on each of the machines that sends the partial results from machines with positive numbers to the machine number 0.

```
  if (MyNodeId() > 0) {
    PutInt(0, divisors_num);
    Send(0);
  } else {  // MyNodeId == 0
    for (int node = 1; node < NumberOfNodes(); ++node) {
      Receive(node);
      divisors_num += GetInt(node);
    }
    cout << divisors_num << endl;
  }
}
```

The final solution works in $O(\sqrt{n}\,/\,M)$ time, i.e., this is the maximum of the time complexities of the instances running on each of the machines.

3. Task "Workshop" (2014)

During an algorithmic workshop students are sitting in a circle and solving problems. Whenever someone comes up with a solution to a problem, he or she shares the idea with his or her two neighbours (which takes exactly one minute), and then they pass it to their neighbours (which also takes exactly one minute), and so on.

If Johny comes up with a brilliant solution at quarter to twelve, what time will Chris hear about it? How many minutes does it take for a solution to reach from Kate to Tom? That is the kind of queries your program has to answer.

Input

The input data is provided by an interactive library. Your program can call six functions from the library:

- `int NumberOfStudents()`: returns the number n of students participating in the workshop ($3 \leq n \leq 10^9$). The students are numbered with consecutive integers from 1 to n.
- `int FirstNeighbour(int i)`: returns the number of the *first neighbour* of the i-th student ($1 \leq i \leq n$). The students have a hard time distinguishing left from right, therefore they prefer to call their neighbours in the order of their numbers. That is, the *first neighbour* of a given student always has a number smaller than his of her *second neighbour*.
- `int SecondNeighbour(int i)`: returns the number of the *second neighbour* of the i-th student ($1 \leq i \leq n$).
- `int NumberOfQueries()`: returns the number m of queries your program has to answer ($0 \leq m \leq 200$). The queries are numbered with consecutive integers from 1 to m.
- `int QueryFrom(int i)`: for the i-th query ($1 \leq i \leq m$), returns the number of the student who came up with a solution.
- `int QueryTo(int i)`: for the i-th query ($1 \leq i \leq m$), returns the number of the student willing to know when he or she is going to hear the solution.

Output

Your program should print exactly m lines to the standard output. The i-th line should contain the answer to the i-th query, i.e., the number of minutes it takes for a solution to reach from one student to the other.

3.1. *Solution*

In the problem a cycle is specified using an oracle which for a given vertex returns its two neighbours. The neighbours are returned in an order that is not necessarily consistent with the order of the vertices on the cycle. A number of queries are given, each consisting of two vertices, and the task is to compute for each query the length of the shortest path between the two vertices.

The first step of the model solution is to select a subset of vertices, which we call *checkpoints*. The checkpoints include all the vertices which are part of any query and additionally some number of randomly selected vertices. Later we discuss how to choose this number. After the checkpoints have been selected, each checkpoint is randomly assigned to some machine (node). All the random choices are made with a deterministic pseudorandom number generator so that all machines select the same checkpoints without needing to communicate.

In the second step each machine processes the checkpoints assigned to it. Starting from a checkpoint the process running on the machine traverses the cycle in both directions until it reaches (at both ends) any other checkpoints (they might be assigned to a different machine). While traversing the cycle, the process counts the number of visited vertices. Finally, it sends to the first machine a list of statements of the following form: the distance between the checkpoints a and b equals d and there is no other checkpoint between them.

The third step is run only on the first machine. First, it receives messages from all the other machines. Using information from the messages, the machine computes the order of checkpoints on the cycle and the distance between each two neighboring checkpoints. After linear-time processing of this information, it is easy to answer each query in constant time.

We are left with the problem of choosing the number of checkpoints. Recall that M denotes the number of machines. Each machine processes a random fraction of $\frac{1}{M}$ of the checkpoints which makes the expected running time of each machine $O(\frac{n}{M})$. However, from a theoretical point of view, this kind of a statement is worthless. Consider an imaginary situation in which we need to perform computations that take $\Theta(T)$ total time, and we pick a random machine to perform all of it. Then each machine spends $\Theta(T)$ time with probability $\frac{1}{M}$, which gives exactly $\Theta(T/M)$ expected time for each machine, just like in the case of our solution. At the same time, we are interested in the running time of the *slowest* machine, which is clearly still $\Theta(T)$.

For this reason, we study the performance of our solution experimentally. Luckily, the running time depends mostly on the size of the input data, not on its structure. Our simulations show that with M randomly selected checkpoints, the longest running machine uses $O(\frac{n \lg M}{M})$ time, compared to $O(\frac{n}{M})$ expected time, which is consistent with a theoretical analysis in (David and Nagaraja, 2003, p. 135). It is possible to reduce the variation between machines by increasing the number of checkpoints. In practice, our solution which always selects $10\ 000$ checkpoints is about 1.5–2 times faster than the one that selects M checkpoints.

3.2. *Tests*

It seems that virtually any nontrivial test is sufficient to distinguish solutions based on an incorrect algorithm. However, a bit more care is required to distinguish solutions that are correct but may be too slow – e.g. a variant of the model solution that selects as the checkpoints only the queries endpoints and the vertices $1, 2, \dots, M$ instead of randomly selected vertices. To make such solutions exceed the time limit we need to pay attention to keep large contiguous fragments of the cycle without any query vertex and leave a large fragment of the initial $1, 2, \dots, n$ cycle around the vertex 1 unaltered. The remaining part of the cycle is permuted either by performing a circular shift on the binary representations of vertex numbers or by xor-ing them with some fixed number. This method allows $O(1)$-time calculation of vertex neighbours and produces a cycle looking sufficiently random to make it difficult to come up with a clever incorrect solution exploiting this particular structure of the test.

4. Task “Assistant” (2014)

The life of an assistant is not easy. Not only did the professor order him to write a terribly long review, she also requested some corrections today.

Pushing keys of a keyboard is very tiring, so the primary goal of the assistant is to push keys as few times as possible, while correcting the review. The keyboard that he is using with a single click allows him to delete a character in the review, change a character to a different one or insert one character anywhere in the review.

To make things worse, the assistant has a very peculiar sense of esthetics. He likes letters from the beginning of the alphabet (like `a`, `b` or `c`), but is disgusted by the letters from the end of the alphabet (in particular, `y` and `z`). Each time he presses a key and changes a letter that comes earlier in the alphabet to a letter that comes later (for example, `m` to `p`), he suffers an esthetic shock, which is devastating for him. Because of that, the secondary goal of the assistant is to minimize the number of such changes.

Input

The first line of the standard input contains two integers k and l ($1 \le k,\ l \le 100\ 000$), that specify the lengths of the first and the second version of the review. The following two lines contain the two versions themselves. Each review consists only of lowercase letters.

Output

Your program should output a single line containing the minimal number of keyboard presses that the assistant has to perform in order to correct the review, followed by the minimal number of esthetic shocks that he will suffer.

4.1. *Solution*

The problem considered in this task is a variant of the well-known *edit distance* problem. Our solution will refer to the classical dynamic programming approach to this problem, which has been described in a number of textbooks (see, e.g., Cormen *et al.*, 2009). The solution that we obtain can be easily extended with minimizing the number of esthetic shocks. In short words, it suffices to, instead of storing only the edit distances, store integer pairs that describe the edit distance and the number of esthetic shocks, and compare them lexicographically.

Denote by $a_1, \ldots, a_k$ the characters in the first version of the review and by $b_1, \ldots, b_l$ the characters in the second version. Our goal is to compute a two dimensional $k \times l$ matrix D, where $D(i, j)$ contains the minimum number of changes that the assistant has to perform in order to change $a_1, \ldots, a_i$ into $b_1, \ldots, b_j$. Just like in the edit distance problem, the values $D(1, \cdot)$ and $D(\cdot, 1)$ can be computed in a straightforward way, whereas for $2 \le i \le k$ and $2 \le j \le l$, $D(i, j)$ can be computed in constant time, given $D(i-1, j)$, $D(i, j-1)$ and $D(i-1, j-1)$.

Assume that the topmost row of matrix D contains elements $D(1, \cdot)$ and the leftmost column contains $D(\cdot, 1)$. Partition the matrix into M *stripes* consisting of l/M consecutive columns (for simplicity, we assume that l is divisible by M), where M is the number of machines available. Each machine is responsible for filling in the entries of D in one stripe. Let the i-th machine (for $i \in \{1, \ldots, M\}$) be responsible for the i-th stripe from the left. See Fig. 1 for illustration.

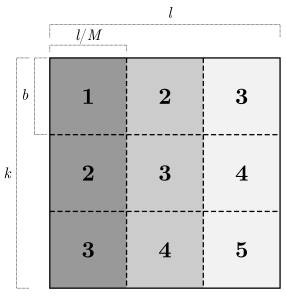


Fig. 1. The process of computing the matrix D. Stripes assigned to different machines have been marked with different shades of gray. The numbers inside the matrix specify the phase number, when the respective part of the matrix is computed (see below).

The first machine fills in the first (leftmost) stripe, starting from the topmost row. Consider the rightmost column in the first stripe. Observe that the contents of this column is everything the second machine needs to know, in order to fill in its stripe. The entries in the rightmost column of the first stripe are filled in by the first machine from top to bottom, and as they are being computed, the first machine sends them to the second machine. The second machine then fills in its stripe and sends the contents of the cells in the rightmost column of its stripe to the third machine, and so on.

The correctness of this approach should be clear. However, we need to improve it a little bit, in order to make it efficient. Clearly, each machine requires $O(kl/M)$ time to fill in its cells. However, all machines (except for the last one) send k messages, each containing a single number. This may be very inefficient, but can be fixed easily, as a machine may send the contents of cells in batches, each containing b numbers, thus reducing the number of messages to $\lceil k/b \rceil$. This obviously does not impact the running time of each machine.

However, there is one more efficiency aspect that we should take care of. Namely, we need to assure that the machines do not wait long for the numbers they need to have in order to perform computation. If each machine sends the contents of the rightmost column *after* filling in its entire stripe (i.e. sends batches of $b = k$ messages), then our solution becomes essentially sequential. On the other hand, we know that $b = 1$ is also not a good choice, for performance reasons.

Let us analyze how to pick a good value of b. For the analysis, assume that all the machines are perfectly synchronized, that is, in each time unit a machine can fill in exactly one cell of its stripe (or wait for the data it needs to continue working). Consider now a phase of exactly bl/M time units. Since each stripe has width l/M, the first machine sends the first message to the second machine exactly after the first phase. In the second phase, the second machine starts working (fills in the first b rows of its stripe) while the first one keeps on working on the following rows. From the second phase, the second machine no longer needs to wait, as it receives the necessary data exactly the moment when it needs it. In general, the i-th machine starts to work in the i-th phase after $(i-1)bl/M = O(ibl/M)$ time units. Thus, the M-th machine starts after $O(bl)$ time units and then works for $O(kl/M)$ time units. Hence, all the machines finish within $O(bl + kl/M)$ time units and send $O(k/b)$ messages each.

By setting $b = \lfloor k/M \rfloor$ we assure that the waiting time is dominated by the computing time, which means that our solution parallelizes the single-machine solution in a perfect way. At the same time, the total number of messages sent is moderate ($O(M^2)$).

5. Task “Sabotage” (2015)

The city of Megabyteopolis was built upon a large lake and consists of a number of isles connected with bridges. The bridges may run above other bridges.

A group of saboteurs wants the current president Byteasar not to be reelected. They plan to impact the public opinion by exposing Byteasar’s administration’s helplessness in the case of a major emergency. Specifically, they decided to blow up one of the bridges (they cannot afford blowing up more). The sabotage could be considered successful only if there was no other way between the isles previously connected by the destroyed bridge. Your task is to find the number of bridges that the saboteurs should consider when working out the details.

Input

- `int NumberOfIsles()`: returns n ($1 \le n \le 200\,000$) – the number of isles constituting the city of Megabyteopolis. The isles are numbered 0 through $n-1$.
- `int NumberOfBridges()`: returns m ($1 \le m \le 10^8$) – the number of bridges in the city. The bridges are numbered 0 through $m-1$.
- `int BridgeIntA(int i)`: returns the first isle connected by the bridge i.
- `int BridgeIntB(int i)`: returns the second isle connected by the bridge i.

Output

The output should contain a single integer – the number of bridges whose blowup could result in the sabotage being considered successful.

5.1. *Solution*

In this task we are asked to solve a basic graph problem: for a given undirected graph $G = (V, E)$ we need to compute the number of *bridges*. A bridge is defined here as an edge of G whose removal results in an increase of the number of connected components of G. Denote by $B(G)$ the set of bridges of G. A textbook algorithm (e.g., Sedgewick, 2002) for computing $B(G)$ is based on an extension of the *depth-first search* (DFS) algorithm and runs in $O(n+m)$ time. The number of vertices in our graph is quite small, i.e., the bound on the order of 10^5 is typical for graph tasks even in the traditional, non-distributed setting. The number of edges m in our case can be, however, much larger.

Unfortunately, DFS is not an algorithm that can be parallelized easily. Nevertheless, we do not need to entirely abandon the idea of using DFS: our strategy is to use the multiple machines to reduce our problem instance to an instance with only $O(n)$ edges. In such a reduced instance, we use DFS to find the bridges.

Definition 1. *Consider a graph $H = (V, E)$. We define a* bridge certificate *of H to be a set $X \subseteq E$ such that for any $Y \subseteq V \times V$, $B((V, E \cup Y)) = B((V, X \cup Y))$.*

As a result, replacing a subset of edges of G with its bridge certificate does not affect the set of bridges of G. We are going to use the bridge certificates to detect and remove edges of G. The following lemma describes a construction of a bridge certificate. For completeness we give its proof in the Appendix.

Lemma 1. *Let $H = (V, E)$ and $n = |V|$, $m = |E|$. Then there exists a bridge certificate X of H such that $|X| \leq 2n$ which can be computed in $O(n+m)$ time.*

By Lemma 1, we can take any subset E' of edges of G, find a bridge certificate X of (V, E') and replace E' in G with $X \subseteq E'$ in $O(n+|E'|)$ time. We call this step a *reduction* with respect to E'.

It turns out that we can easily perform the reductions in a parallel fashion. For simplicity, first assume that we have only two machines, i.e., $M = 2$. We partition the edge set E into two sets E_0, E_1 of roughly equal size. The machine i, for $i = 0, 1$, performs the reduction step on the set E_i, obtaining a certificate X_i of size at most $2n$, in $O(n+|E_i|)$ time. Next, machine 1 sends the set X_1 to machine 0. In the last step, machine 0 runs DFS to find the set $B((V, X_0 \cup X_1))$. This, however, takes only $O(n)$ time, as $|X_0 \cup X_1| \leq 4n$. By the definition of a certificate,

$$B(G) = B((V, E_0 \cup E_1)) = B((V, X_0 \cup E_1)) = B((V, X_0 \cup X_1)).$$

This concludes that indeed this approach finds all the bridges of G.

In order to develop a distributed algorithm using $M > 2$ machines, we perform multiple reduction phases. In the first phase, the set of edges is partitioned among the M machines, and each machine computes a certificate of the edges assigned to it. In the following phases the certificates are merged in pairs: two certificates produced in the previous phase are sent to a machine that takes their union and computes the certificate of the resulting graph.

Let us describe this process formally. Assuming that the machines are numbered 0 through $M-1$, we split the input edge set E arbitrarily into M parts $E_0, \ldots, E_{M-1}$, each of size $O(m/M)$. Our distributed algorithm runs in $K = \lceil \log_2 M \rceil + 1$ phases numbered 0 through $K-1$. In the k-th phase ($k = 0, \ldots, K-1$) only the machines with identifiers j divisible by 2^k are active and actually do perform some work. With each active machine j we associate two sets Y_j, X_j (Y_j, $X_j \subseteq E$) whose contents depend on the phase number k. Before the k-th phase:

- Y_j is a bridge certificate of the graph

$$H_j = (V, E_j \cup E_{j+1} \cup \ldots \cup E_{j+2^k-1}).$$

  In the above we set $E_{j+l} = \emptyset$ if $j + l \geq M$.
- If $k = 0$ then $Y_j = E_j$. Otherwise, $|Y_j| \leq 4n$.

After the k-th phase the set X_j is a bridge certificate of H_j and $|X_j| \leq 2n$. Note that it follows that after the phase $K-1$, X_0 is a bridge certificate of G and $|X_0| = O(n)$.

At that point the machine 0 runs DFS on (V, X_0) to compute the set of bridges of G in $O(n)$ time.

It remains to show how to implement the phases so that the invariants imposed on the sets Y_j, X_j are satisfied. Before the first phase we set $Y_j = E_j$. Assume that the phase $k - 1$ has been completed. We perform a reduction of Lemma 1 on the set Y_j in order to obtain the set X_j. This takes $O(n + |Y_j|)$ time, which is $O(n)$ for $k > 0$ and $O(n + m/M)$ if $k = 0$. The last step is to initialize the sets Y_j before the next, $(k + 1)$-th phase. To do that, for each j divisible by 2^{k+1}, we set $Y_j = X_j \cup X_{j+2^k}$ if $j + 2^k < M$ and $Y_j = X_j$ otherwise. As for all l, $|X_l| \leq 2n$, clearly we now have $|Y_j| \leq 4n$. To implement this step, the machine $j + 2^k$ sends the entire set X_{j+2^k} to the machine j. This requires a single message of $O(n)$ bytes. Fig. 2 depicts the phases of our distributed algorithm.

In each phase every machine remains idle or sends $O(n)$ bytes, or receives $O(n)$ bytes. Consequently, the first phase takes $O(n + m/M)$ time on each machine and each of the $K - 1$ remaining phases runs in $O(n)$ time on each machine. Thus, the time complexity of this solution is $O(m/M + n \log M)$.

5.2. *Tests*

The library providing the test data had to be robust enough to serve graphs with large edge sets and nontrivial 2-edge-connected components (i.e., connected components of G formed after removing all the bridges), given limited time and space. It seems that a hard test case is a graph with a maximum number of edges and possibly large number of bridges. In such a case at least some of the 2-edge-connected components should be very dense.

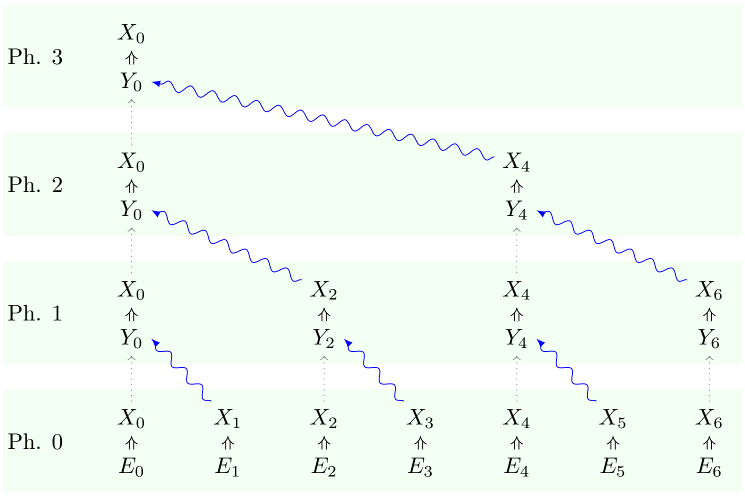


Fig. 2. The phases of the distributed algorithm when $M = 7$. In this case $K = 4$ phases are performed. The blue arrows illustrate the communication between the machines in the corresponding phases.

The infrastructure for generating test graphs provided a general graph interface along with a few specialized implementations (a vertex, a path, a cycle, a clique, a pseudo-random graph, a set of loops) that could serve the edges in $O(1)$ time with constant space consumption, regardless of the graph size. As an example, a cycle on n vertices numbered 0 through $n-1$ can be represented with a single integer n: when asked for the cycle's i-th edge, we just return $(i,\ (i+1) \bmod n)$.

Such graphs could be then combined into larger and more sophisticated graphs by unions and direct sums and also extended by adding specified edges, which were typically used to ensure the desired structural properties of the served graph. For example, this allowed to easily generate a tree of size 100 with each vertex replaced with a random 2-edge-connected graph with 1000 vertices and between 100 000 and 500 000 edges. Such a graph had on the order of 10^7 edges in total, 99 bridges and could be represented with the number of bytes on the order of 10^2. At the highest level, the vertices were assigned random identifiers, whereas the list of edges was randomly permuted.

The size of the in-memory representation of each test cases was $O(n)$ per machine and the sophistication level of the served graphs was limited only by the need to return the requested edge in time on the order of 100ns.

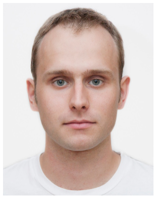
6. Conclusions

We described a novel format of programming competitions, aimed at familiarizing students with an increasingly important area of computer science – design of distributed algorithms. While there is a considerable engineering effort involved in preparing the backend for such a competition, we hope that an increasing number of competitions (maybe including IOI in the future) will feature tracks or problems of a distributed nature, to reflect the industry's shift toward cloud-based and distributed computing.

References

Chávez, L.H. (2015). *libinteractive*: a better way to write interactive tasks. *Olympiads in Informatics*, 9, 3–14.
Cormen, T.H., Leiserson, C.E., Rivest, R.L. Stein, C. (2009), *Introduction to Algorithms*. MIT Press.
David, H.A. Nagaraja, H.N. (2003). *Order Statistics* (3rd Edition). Wiley.
Dean, J. Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM – 50th anniversary issue: 1958–2008*, 51(1), 107–113.
Forišek M. (2013). Pushing the boundary of programming contests. *Olympiads in Informatics*, 7, 23–35.
Fox, A. and Brewer E. (1999). Harvest, yield and scalable tolerant systems. In: *Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99)*. IEEE CS, 174–178.
Komm D. (2011). Teaching the concept of online algorithms. *Olympiads in Informatics*, 5, 58–70.
Kulczyński, T., Łącki, J., Radoszewski, J. (2011). Stimulating students' creativity with tasks solved using precomputation and visualization. *Olympiads in Informatics*, 5, 71–81.
Sedgewick, R. (2002). *Algorithms in C++, Part 5: Graph Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston.

A. Karczmarz (1990), PhD student at Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland, jury member of multiple programming contests organized by University of Warsaw, coorganizer of Algorithmic Engagements 2015. In his research he focuses on algorithms and data structures.



J. Łącki (1986), postdoctoral researcher at Sapienza University of Rome, Italy, IOI Scientific Committee elected member, appointed chair for 2016, responsible for task selection at Algorithmic Engagements for many years, former head organizer of Polish Training Camp. In his research he focuses on graph algorithms.

A. Polak (1991), PhD student at Department of Theoretical Computer Science, Faculty of Mathematics and Computer Science, Jagiellonian University, Kraków, Poland, judge at the ACM Central Europe Regional Contest in 2012, 2013, and 2014. His research interests lie in algorithms, complexity theory, and computer vision.

J. Radoszewski (1984), assistant professor at Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland, and Newton International Fellow at King's College London, UK, chair of the jury of Polish Olympiad in Informatics, co–chair of the Scientific Committee of CEOI'2011 in Gdynia, former member of Host Scientific Committees of IOI'2005, CEOI'2004, BOI'2008, and BOI'2015. His research interests focus on text algorithms and combinatorics.

J.O. Wojtaszczyk (1980), Staff Software Engineer at Google, Warsaw, judge at the ACM ICPC World Finals in 2011, 2012, 2013, 2015 and 2016, coorganizer of the Google Code Jam since 2012, main organizer of the Distributed Code Jam. The primary focus of his engineering work is around cluster management.

Appendix: Proof of Lemma 1

Lemma 1. *Let $H = (V, E)$ and $n = |V|$, $m = |E|$. Then there exists a bridge certificate X of H such that $|X| \le 2n$ which can be computed in $O(n+m)$ time.*

Proof. We compute the set X in the following way. First, compute some spanning forest F (we identify it with a set of its edges) of H using any graph search algorithm. This takes $O(n+m)$ time. Then, compute some spanning forest F' of $H' = (V, E \setminus F)$. Finally, set $X = F \cup F'$. Clearly, $|X| \le 2n$ and $X \subseteq E$.

We now prove that X is indeed a bridge certificate of H. Let $Y \subseteq V \times V$. First, let us show that the graphs $H_1 = (V, E \cup Y)$ and $H_2 = (V, X \cup Y)$ have the same connected components. Clearly, if there exists a path $u \rightsquigarrow v$ in H_2 then a path $u \rightsquigarrow v$ exists in H_1, as H_2 is a subgraph of H_1. Conversely, if u and v are connected in H_1 by a path P, then any edge $(a, b) \in P \setminus Y$ such that $(a, b) \in E \setminus X$ can be replaced by a path $a \rightsquigarrow b$ contained entirely in F (recall that F is a spanning forest of H).

For a graph G, define $G - e$ as G with the edge e removed. Now assume that (u, v) is a bridge in H_1, i.e., $(u, v) \in B(H_1)$. Then, there is no path $u \rightsquigarrow v$ in $H_1 - (u, v)$. As $H_2 - (u, v)$ is a subgraph of $H_1 - (u, v)$, there is also no path $u \rightsquigarrow v$ in $H_2 - (u, v)$. Moreover, H_1 and H_2 have the same connected components and thus $(u, v) \in X \cup Y$. Thus, $(u, v) \in B(H_2)$ and consequently $B(H_1) \subseteq B(H_2)$.

Finally, suppose that $(u, v) \in B(H_2)$. Then, u and v are connected in H_2, but no path $u \rightsquigarrow v$ exists in $H_2 - (u, v)$. As H_2 is a subgraph of H_1, $(u, v) \in E \cup Y$. Let us show that no path $u \rightsquigarrow v$ exists in $H_1 - (u, v)$. Assume the contrary and let P be such a path. If $P \subseteq X \cup Y$, then P would be a $u \rightsquigarrow v$ path in $H_2 - (u, v)$, which is not possible. Hence, there exists some edge $(a, b) \in P$ such that $(a, b) \in E \setminus X \setminus \{(u, v)\}$. As $(a, b) \notin F$ and $(a, b) \notin F'$, there exist paths Q and Q' from a to b in both spanning forests F and F', correspondingly. At least one of these paths, say Q, does not contain (u, v). We may replace the edge (a, b) with the path $Q \subseteq X$. By replacing all such edges (a, b) with paths contained in X, we obtain a path $u \rightsquigarrow v$ in $H_2 - (u, v)$, a contradiction. Thus, $(u, v) \in B(H_1)$ and, consequently, $B(H_2) \subseteq B(H_1)$.