DSRC

DNA Sequence Reads Compression

User Manual

Version 1.0, May 1, 2012

Contents

1 Introduction

1.1 What is DSRC?

DNA Sequence Reads Compression is an application designed for lossless compression of DNA sequencing reads stored in FASTQ format. The used compression algorithm details were described in research paper: S. Deorowicz and Sz. Grabowski, "Compression of DNA sequence reads in FASTQ format", *Bioinformatics* 27(6):860–862 (2011).

1.2 Main features

- Effective compression of whole DNA sequencing data stored in FASTQ format.
- Decompression of whole archive or single records extraction with random access.
- Support for DNA reads stored in SOLEXA, SOLiD and LS454 type formats with variable sequence lengths.
- Easy integration with Python and C++ programs.
- Available both for Linux and Windows 64-bit operating systems.
- Open source C++ code under GNU GPL 2 license.

1.3 Compression factor and speed

In terms of lossless compression factor (the ability to reduce the file size), DSRC is usually 35–55% better than gzip and 15–25% better than bzip2, which are currently the most popular methods of FASTQ files compression methods. As for compression speed, DSRC is about 5 times faster than gzip and bzip2, while in decompression—DSRC is about 2.5 times slower than gzip, but also about 2 times faster than bzip2. The achieved compression factor and speed were presented in aforementioned research paper.

1.4 Contact and support

- Official website: `http://sun.aei.polsl.pl/dsrc/index.html`
- Contact and bug reports: `sebastian.deorowicz[at]polsl.pl`, `lucas.roguski[at]gmail.com`

2

Quick start

2.1 Download

The compiled binaries with Python module for both Linux and Windows operating systems can be downloaded from official website: `http://sun.aei.polsl.pl/dsrc/index.html`.

The Windows x64 binary was compiled using msvc 10.0 and linux x64 binary was compiled using gcc 4.6.1 both in static runtime variants.

The Python module was compiled in shared variant, so boost::python runtime library is required to run properly.

2.2 Building

DSRC should compile on both Windows and Linux platforms.

To compile DSRC on Linux platform there are provided three makefiles, each for different compiler: g++, Intel icpc or AMD Open64.

To compile on Windows platform use project solutions dsrc-vs2k10.sln (or dsrc-vs2k8.sln) for Microsoft Visual Studio 2010 (or 2008) toolbox, although the 2010 version is highly recommended. DSRC can be also compiled using Cygwin (32-bit) or MinGW-W64 (64-bit) toolbox using linux makefiles.

To build the DSRC Python module the boost::Python library is needed, which can be downloaded from official boost website http://www.boost.org/. The Python module can be build using the Boost Build tool bjam, where the different Jamroot files were provided for each compiler toolset.

2.3 Program usage

Main application can be run from the command prompt:

dsrc <*mode*> [options] <*input_file_name*> <*output_file_name*>

with available modes:

- **c** — compression,

- **d** — decompression,
- **e** — extraction.

Available compression options:

- **-c** — Perform CRC32 check after compressing to check data consistency.
- **-l** — Enable LZ–matching to achieve better compression factor (about twice slower compression speed).
- **-lm<*n*>** — Use at most *n* MB of memory for finding LZ–matches. Default memory size (MB): 1024, min: 64, max: 65536.

Available extraction options:

- **-r<*n*>** — extract single record of number *n*.

Usage examples:

- Compress SRR001471.fastq file:
  `dsrc c SRR001471.fastq SRR001471.dsrc`
- Compress SRR001471.fastq file with CRC32 checking:
  `dsrc e -c SRR001471.fastq SRR001471.dsrc`
- Compress SRR001471.fastq file with LZ–matching with using at most 4096 MB of memory for LZ–matches storage:
  `dsrc e -l -lm4096 SRR001471.fastq SRR001471.dsrc`
- Decompress the full SRR001471.dsrc archive:
  `dsrc d SRR001471.dsrc SRR001471.out.fastq`
- Extract only record no. 532 from SRR001471.dsrc archive:
  `dsrc e -r532 SRR001471.dsrc SRR001471.out.fastq`

3

DSRC integration

DSRC can be integrated with your program written in C++ or Python. We provide C++ library and Python module with very similar available functionality. Although the method and member names are almost identical in both cases, for clarity the C++ and Python descriptions are divided into separate subsections.

3.1 Python API

To start using the compressor functionality in Python you just only need to import **pydsrc** module in your project.

In order to provide high performance of compression routines the core operations were written in C++ and exported to Python. This requires introducing own data array structure *array_uc8* with analogous interface as Python *list* for holding sequence reads and qualities (*unsigned char*) with additional conversion functions. To convert *array_uc8* object to Python *list* use the **to_list_uc8** function and analogously to convert Python *list* object to *array_uc8* object — function **to_array_uc8**.

3.1.1 FastqRecord

FastqRecord is the basic structure holding the single DNA sequencing read information. The string values are stored as an *unsigned char* and the whole string is represented as *list_uc8*. The key members are **Title**, **Sequence**, **Plus**, and **Quality**.

3.1.2 FastqFile

FastqFile is a class for FASTQ records data representation. It handles I/O operations with buffering. It also enables CRC32 hash computation on reading or writing new records from/to file. Table 3.1 shows **FastqFile** available public methods presented. Public properties are presented at Table 3.2. All methods throw an exception on error or failure.

3.1.3 DsrcFile

DsrcFile is a class representing DSRC file with compression routines, which handles I/O operations with buffering. It enables compression, decompression, and extraction of records with additionally setting LZ compression parameters. Table 3.3 showsh **FastqFile** available public methods — all methods throw an exception on error or failure. Table 3.4 presents public properties of the class. It is important to note, that properties cannot be set, while **DsrcFile** is already processing an archive — when compression, decompression, or extraction routines have been stared.

Table 3.1: FastqFile public methods

Method	Returns	Parameters	Description
Open	—	string filename	Open specified file.
Create	—	string filename	Create a new file or overwrites if one exists.
Close	—	—	Close the file.
ReadNextRecord	boolean	FastqRecord rec	Get the next buffered record. Returns True on succesful read and False on EOF.
WriteRecord	—	FastqRecord rec	Write the record.

Table 3.2: FastqFile properties

Property	Modifier	Type	Description
Size	Read-only	int	Size of file.
Position	Read-only	int	Current position of file pointer.
Eof	Read-only	boolean	End of file reach indicator.
Crc32Checking	R/W	boolean	Use CRC32 checking. Default: False.
RecordsCrc32Hash	Read-only	int	Records' data CRC32 hash.
FileCrc32Hash	Read-only	int	File's CRC32 hash.

Table 3.3: DsrcFile public methods

Method	Returns	Parameters	Description
StartCompress	—	string filename	Create a new DSRC archive and prepares for compression.
WriteRecord	—	FastqRecord rec	Write a new record to file.
FinishCompress	—	—	Finalize the compression of the archive and performs cleanup.
StartDecompress	—	string filename	Open a DSRC archive and prepares for decompression.
ReadNextRecord	boolean	FastqRecord rec	Read the next decompressed FASTQ record from archive. Returns TRUE on success and FALSE on EOF or error.
FinishDecompress	—	—	Finalize the decompression of the archive and performs cleanup.
StartExtract	—	string filename	Open a DSRC archive in random-access pattern and prepares for extraction.
ExtractRecord	—	FastqRecord rec, uint64 id	Extract the decompressed FASTQ record with specified ID from archive.
FinishExtract	—	—	Finalize the extraction and performs cleanup.
Reset	—	—	Reset the file on error and performs cleanup.

Table 3.4: DsrcFile properties

Property	**Modifier**	**Type**	**Description**
Size	Read-only	int	Size of file.
Position	Read-only	int	Current position of file pointer.
LzMatching	R/W	boolean	Use LZ-matching to improve compression. Default: false.
LzMemorySize	R/W	int	Maximum size of memory in MB used for LZ-matching. Available size: $128 - 65536$, where size must be power of 2. Default: 1024.

Table 3.5: Compressor public methods

Method	**Returns**	**Parameters**	**Description**
Compress	—	string fastqFilename, string dsrcFilename	Compress the FASTQ file to DSRC archive.
Decompress	—	string dsrcFilename, string fastqFilename	Decompress the DSRC archive to file in FASTQ format.
Extract	—	string dsrcFilename, string fastqFilename, int recId	Extract single record from DSRC archive saving results to file in FASTQ format.

3.1.4 Compressor

Compressor is a class providing automated compression routines, reducing the needed work and interaction for user. Enables compression, decompression and extraction of records with additionally setting CRC32 checking and LZ compression parameters. Table 3.5 shows **Compressor** available public methods, while public properties are presented at Table 3.6. All methods throw an exception on error or failure.

Table 3.6: Compressor properties

Property	**Modifier**	**Type**	**Description**
VerboseLevel	R/W	VerboseLevel	Level of output messages. Default: VERBOSE_INFO.
LzMatching	R/W	boolean	Use LZ-matching to improve compression, but almost twice the slower compression time. Default: false.
LzMemorySize	R/W	int	Maximum size of memory in MB used for LZ-matching. Available size: 128–65536, where size must be power of 2. Default: 1024.
Crc32Checking	R/W	boolean	Use CRC32 checking. Default: false.

3.2 Python examples

3.2.1 Using automated Compressor module

```python
from pydsrc import *

filename_in = "SRR001471.fastq"
filename_dsrc = "SRR001471.dsrc"
filename_test = "SRR001471.test.fastq"
filename_ext = "SRR001471.ext.fastq"

# create and configure DSRC compressor
dsrc = Compressor()

# show only error messages
dsrc.VerboseLevel = VERBOSE_ERRORS

# enable Lz-Matching with max. of 4096 MB memory
dsrc.LzMatching = True
dsrc.LzMemorySize = 4096

# enable CRC32 control checksum check
dsrc.Crc32Checking = True

# compress 'filename_in' FASTQ file and save the DSRC archive as 'filename_dsrc'
dsrc.Compress(filename_in, filename_dsrc)

# decompress 'filename_dsrc' archive and save data to 'filename_test' FASTQ file
dsrc.Decompress(filename_dsrc, filename_test)

# extract record of number '40' from 'filename_dsrc' DSRC archive
# saving record to 'filename_ext' file in FASTQ format
dsrc.ExtractRecord(filename_dsrc, filename_ext, 40)
```

3.2.2 Manual compression using DsrcFile and FastqFile

```python
from pydsrc import *
from time import clock

filename_in = "SRR001471.fastq"
filename_out = "SRR001471.dsrc"
t0 = clock()

# create and configure dsrc archive
dsrc = DsrcFile()
dsrc.LzMatching = True
dsrc.LzMemorySize = 4096

# open fastq file
fastq = FastqFile()
fastq.Open(filename_in)

# start compression
dsrc.StartCompress(filename_out)
print "Compressing", filename_in, "of␣size", fastq.Size

# read all records from fastq file and write them to dsrc archive
```

```
rec = FastqRecord()
while fastq.ReadNextRecord(rec):
    dsrc.WriteRecord(rec)

# finish compression of dsrc archive and close file
dsrc.FinishCompress()

# close fastq file
fastq.Close()

# print some results ;)
ds_size = dsrc.Position
t = (clock() - t0)
if t > 0.0:
    s = fastq.Size / t / 1000000.0
    print "Speed:\t\t\t", "%.2f"%s, "MB/s"
if ds_size > 0:
    print "Comp.␣factor:\t\t", "%.2f"%(1.0*fastq.Size/ds_size)
print "Processing␣time:\t", "%.1f"%t, 's'
```

3.2.3 Manual decompression using DsrcFile and FastqFile

```
from pydsrc import *
from time import clock

filename_in = "SRR001471.dsrc"
filename_out = "SRR001471.result.fastq"
t0 = clock()

# start decompression of dsrc archive
dsrc = DsrcFile()
dsrc.StartDecompress(filename_in)

# create output fastq file
fastq = FastqFile()
fastq.Create(filename_out)
print "Decompressing", filename_in

# read all records from archive and write them to fastq file
rec = FastqRecord()
while dsrc.ReadNextRecord(rec):
    fastq.WriteRecord(rec)

# finish decompression of dsrc archive and close file
dsrc.FinishDecompress()

# flush and close output file
fastq.Close()

# print some results ;)
t = (clock() - t0)
fq_size = fastq.Position
if t > 0.0:
    s = fq_size / t / 1000000.0
    print "Speed:\t\t\t", "%.2f"%s, "MB/s"
if dsrc.Size > 0:
    print "Comp.␣factor:\t\t", "%.2f"%(1.0*fq_size/dsrc.Size)
print "Processing␣time:\t", "%.1f"%t, 's'
```

3.2.4 Manual extraction of records using DsrcFile and FastqFile

```
from pydsrc import *
from time import clock

filename_in = "SRR001471.dsrc"
filename_out = "SRR001471.ext.fastq"
t0 = clock()

# start extraction of dsrc archive
dsrc = DsrcFile()
dsrc.StartExtract(filename_in)

# create output fastq file
fastq = FastqFile()
fastq.Create(filename_out)

rec = FastqRecord()
# using psuedo random numbers ;)

for id in [2124, 18000, 2126, 10]:
    dsrc.ExtractRecord(rec, id)
    fastq.WriteRecord(rec)

# finish extraction and close archive
dsrc.FinishExtract()

# flush and close output file
fastq.Close()

t = (clock() - t0)
print "Processing␣time:␣", "%.1f"%t, 's'
```

3.3 C++ API

To start using the compressor functionality in C++ you just need to include **dsrc.h** header file and link your application with **libdsrc** library. To reduce the C++ language performance overhead the polymorphisms, exceptions, and RTTI mechanisms were not used.

In DSRC some basic type definitions were introduced, which are presented in Table 3.7.

Table 3.7: C++ basic types introduced in DSRC

Type name	**Full type**
uchar	unsigned char
int32	int
uint32	unsigned int
int64	long long int
uint64	unsigned long long int

3.3.1 FastqRecord

FastqRecord is the basic structure holding the single DNA sequencing read information. The string values are stored as a raw tables of *unsigned char* values — those are: **title** (or sequence header), **sequence**, **plus** and **quality** fields. To each field corresponds a unsigned int variable indicating the length of string sequence — **title_len**, **sequence_len**, **plus_len**, **quality_len** — and allocated size of string — **title_size**, **sequence_size**, **plus_size**, **quality_size**.

3.3.2 FastqFile

FastqFile is a class for FASTQ records data representation. It handles I/O operations with buffering. It also enables CRC32 hash computation on reading or writing new records from/to file. Table 3.8 shows **FasqtFile** public methods. As no exception mechanism was used, most methods return value of *bool* type indicating success **true** or failure/error **false**.

3.3.3 DsrcFile

DsrcFile is a class representing DSRC file with compression routines, which handles I/O operations with buffering. It enables compression, decompression, and extraction of records with additionally setting LZ compression parameters. Table 3.9 shows DsrcFile public methods. Analogously like in **FastqFile** class, as no exception mechanism was used, most methods return value of *bool* type indicating success **true** or failure/error **false**.

3.3.4 Compressor

Compressor is a class providing automated compression routines, reducing the needed work and interaction for user. It enables compression, decompression, and extraction of records with additionally setting CRC32 checking and LZ compression parameters. Table 3.10 presents **Compressor** available public methods.

Table 3.8: FastqFile public methods

Method	**Returns**	**Parameters**	**Description**
Open	bool	const char* filename	Open specified file for reading.
Create	bool	const char* filename	Create a new file or overwrites if one exists for writing.
Close	bool	—	Close the file.
ReadNextRecord	bool	FastqRecord& rec	Get the next buffered record. Return false on EOF (or reading error).
WriteRecord	bool	const FastqRecord& rec	Write the record.
GetFilePos	uint64	—	Return the current position of the file pointer — number of bytes from beginning of the file.
GetFileSize	uint64	—	Return the size of the opened file. When in writing mode it returns 0 — the value is updated after closing the file.
IsCrc32Checking	bool	—	Return the state of CRC32 hash calculations option enabled.
SetCrc32Checking	—	bool state	Enable or disables CRC32 hash calculations.

Table 3.9: DsrcFile public methods

Method	Returns	Parameters	Description
StartCompress	bool	const char* filename	Create a new DSRC archive and prepares for compression.
WriteRecord	bool	const FastqRecord& rec	Write a new record to file.
FinishCompress	bool	—	Finalize the compression of the archive and performs cleanup.
StartDecompress	bool	const char* filename	Open a DSRC archive and prepares for decompression.
ReadNextRecord	bool	FastqRecord& rec	Read the next decompressed FASTQ record from archive. Returns false on EOF or error.
FinishDecompress	bool	—	Finalize the decompression of the archive and performs cleanup.
StartExtract	bool	const char* filename	Open a DSRC archive in random-access pattern and prepares for extraction.
ExtractRecord	bool	FastqRecord& rec, uint64 id	Extract the decompressed FASTQ record with specified ID from archive.
FinishExtract	bool	—	Finalize the extraction and performs cleanup.
Reset	—	—	Reset the file on error and performs cleanup.
GetFilePos	uint64	—	Return the current position of the file pointer — number of bytes from beginning of the file.
GetFileSize	uint64	—	Return the size of the opened archive. When compressing the return value is 0 and is updated only after finishing compression.
IsLzMatching	bool	—	Return the indicator whether LZ-matching is enabled for compression.
SetLzMatching	bool	bool state	Enable or disables LZ-matching. Default: false.
GetLzMemorySize	uint32	—	Return the size of current available memory for LZ-matching.
SetLzMemorySize	bool	uint32	Set the size of memory in MB used for LZ-matching. Available size: 128–65536, where size must be power of 2. Default: 1024.

Table 3.10: Compressor public methods

Method	Returns	Parameters	Description
Compress	bool	const char* fastqFilename, const char* dsrcFilename	Compress the FASTQ file to DSRC archive.
Decompress	bool	const char* dsrcFilename, const char* fastqFilename	Decompress the DSRC archive to file in FASTQ format.
Extract	bool	const char* dsrcFilename, const char* fastqFilename, int64 recId	Extract single record from DSRC archive saving results to file in FASTQ format.
IsLzMatching	bool	–	Return the indicator whether LZ-matching is enabled for compression.
SetLzMatching	bool	bool state	Enable or disables LZ-matching.
IsCrc32Checking	bool	—	Return the state of CRC32 hash calculations option enabled.
SetCrc32Checking	—	bool state	Enable or disables CRC32 hash calculations.

3.4 C++ EXAMPLES

3.4.1 Compression examples

```
int compress()
{
    const char* inFastqFiles[] = {"data/read1.fastq", "data/read2.fastq"};
    const char* outDsrcFiles[] = {"data/read1.dsrc", "data/read2.dsrc"};
    const char* outDsrcFileJoin = "data/readJ.dsrc";

    // compress files using the easy method - Compressor object
    std::cout << "Example␣1␣:␣compression\n";
    Compressor compressor;

    // set additional parameters:
    // lz compression method and CRC32 checksum checking
    compressor.SetLzMatching(true);
    compressor.SetLzMemorySize(4086);
    compressor.SetCrc32Checking(true);

    // compress files
    if (!compressor.Compress(inFastqFiles[0], outDsrcFiles[0])
     || !compressor.Compress(inFastqFiles[1], outDsrcFiles[1]))
    {
        return -1;
    }

    // compress the files into one using DSRC module object
    //:IMPORTANT: reads need to be in the same format (SOLiD or LS454/SOLEXA)
    DsrcFile dsrcFile;

    // check if files exists and open them for reading
    FastqFile fastqFiles[2];
    if (!fastqFiles[0].Open(inFastqFiles[0]))
    {
        std::cout << "Error␣opening␣input␣file:␣";
        std::cout << inFastqFiles[0] << "\n";
        return -1;
    }
    if (!fastqFiles[1].Open(inFastqFiles[1]))
    {
        fastqFiles[0].Close();
        std::cout << "Error␣opening␣input␣file:␣";
        std::cout << inFastqFiles[1] << "\n";
        return -1;
    }

    // set additional compression parameters
    //:IMPORTANT: this is need to be done before compressing
    dsrcFile.SetLzMatching(true);

    if (!dsrcFile.StartCompress(outDsrcFileJoin))
    {
        std::cout << "Error:␣cannot␣start␣compression\n";
        if (dsrcFile.IsError())
            std::cout << dsrcFile.GetError();
        fastqFiles[0].Close();
        fastqFiles[1].Close();
        return -1;
    }

    // read all records from files
    FastqRecord recordBuffer;
    while (fastqFiles[0].ReadNextRecord(recordBuffer))
        dsrcFile.WriteRecord(recordBuffer);
    fastqFiles[0].Close();

    while (fastqFiles[1].ReadNextRecord(recordBuffer))
        dsrcFile.WriteRecord(recordBuffer);
    dsrcFile.FinishCompress();

    std::cout << "Success!\n";
    return 0;
}
```

3.4.2 Decompression examples

```
int decompress()
{
    const char* inDsrcFiles[] = {"data/read1.dsrc", "data/read2.dsrc"};
    const char* outFastqFiles[] = {"data/read1-out.fastq", "data/read2-out.fastq"};
    const char* outFastqFileJoin = "data/readJ.fastq";

    // decompress files using the easy method - Compressor object
    //
    std::cout << "Example␣2␣:␣decompression\n";
    Compressor compressor;

    // compress files
    if (!compressor.Decompress(inDsrcFiles[1], outFastqFiles[1])
     || !compressor.Decompress(inDsrcFiles[0], outFastqFiles[0]))
    {
        return -1;
    }

    // decompress the files into one using DSRC module object
    DsrcFile dsrcFile;

    // check if we can open DSRC file
    if (!dsrcFile.StartDecompress(inDsrcFiles[0]))
    {
        std::cout << "Error:␣cannot␣start␣decompression␣of␣file:␣" << inDsrcFiles[0] << "\n";
        if (dsrcFile.IsError())
            std::cout << dsrcFile.GetError();
        return -1;
    }

    // check if we can create out file
    FastqFile fastqFile;
    if (!fastqFile.Create(outFastqFileJoin))
    {
        std::cout << "Error␣creating␣output␣file:␣" << outFastqFileJoin << "\n";
        dsrcFile.FinishDecompress();
        return -1;
    }
    // read all records from file
    FastqRecord recordBuffer;
    while (dsrcFile.ReadNextRecord(recordBuffer))
        fastqFile.WriteRecord(recordBuffer);
    // finish decompression of the first file and start next
    dsrcFile.FinishDecompress();

    if (!dsrcFile.StartDecompress(inDsrcFiles[1]))
    {
        std::cout << "Error:␣cannot␣start␣decompression␣of␣file:␣" << inDsrcFiles[1] << "\n";

        if (dsrcFile.IsError())
            std::cout << dsrcFile.GetError();
        fastqFile.Close();
        return -1;
    }

    // read all records from file
    while (dsrcFile.ReadNextRecord(recordBuffer))
        fastqFile.WriteRecord(recordBuffer);
    dsrcFile.FinishDecompress();
    fastqFile.Close();
    std::cout << "Success!\n";
    return 0;
}
```

3.4.3 Extraction examples

```
int extract()
{
    const char* inDsrcFiles[] = {"data/read1.dsrc", "data/read2.dsrc"};
    const char* outFastqFile = "data/read1-ext.fastq";

    const uint32 records_num = 10;
    std::vector<uint64> records;
    for (uint32 i = 0; i < records_num; ++i)
        records.push_back(i*1024);          // sample records' ids

    // extract records using the easy method - Compressor object
    std::cout << "Example␣3␣:␣extraction\n";
    Compressor compressor;
    compressor.ExtractRange(inDsrcFiles[1], outFastqFile, records);

    // extract the records from two files and join them together
    DsrcFile dsrcFile;

    // check if we can open DSRC file
    if (!dsrcFile.StartExtract(inDsrcFiles[1]))
    {
        std::cout << "Error:␣cannot␣start␣extraction␣of␣file:␣" << inDsrcFiles[0] << "\n";
        if (dsrcFile.IsError())
            std::cout << dsrcFile.GetError();
        return -1;
    }

    // check if we can create out file
    FastqFile fastqFile;
    if (!fastqFile.Create(outFastqFile))    // overwrites the original file
    {
        std::cout << "Error␣creating␣output␣file:␣" << outFastqFile << "\n";
        dsrcFile.FinishExtract();
        return -1;
    }

    // read all records from file
    FastqRecord recordBuffer;
    for (uint32 i = 0; i < records_num/2; ++i)
    {
        if (!dsrcFile.ExtractRecord(recordBuffer, records[i]))
            break;
        fastqFile.WriteRecord(recordBuffer);
    }

    // everything went fine?
    if (dsrcFile.IsError())
    {
        std::cout << "Error␣while␣extracting␣records...\n";
        std::cout << dsrcFile.GetError();
        dsrcFile.FinishExtract();
        fastqFile.Close();
        return -1;
    }
    // finish decompression of the first file and start next
    dsrcFile.FinishExtract();

    if (!dsrcFile.StartExtract(inDsrcFiles[1]))
    {
        std::cout << "Error:␣cannot␣start␣decompression␣of␣file:␣" << inDsrcFiles[1] << "\n";

        if (dsrcFile.IsError())
            std::cout << dsrcFile.GetError();

        fastqFile.Close();
        return -1;
    }

    // read all records from file
    for (uint32 i = records_num/2; i < records_num; ++i)
    {
        if (!dsrcFile.ExtractRecord(recordBuffer, records[i]))
            break;
        fastqFile.WriteRecord(recordBuffer);
    }
    dsrcFile.FinishExtract();
    fastqFile.Close();

    // everything went fine?
    if (dsrcFile.IsError())
    {
        std::cout << "Error␣while␣extracting␣records...\n";
        std::cout << dsrcFile.GetError();
        return -1;
    }
    std::cout << "Success!\n";
    return 0;
}
```