



95.12 - ALGORITMOS Y PROGRAMACION II

Trabajo práctico 0: programación C++

2^{do} CUATRIMESTRE DE 2020

Alumnos:

| | | |
|-------------------|-------|--------------------|
| Burna Lucas | 99608 | lburna@fi.uba.ar |
| Perczyk Francisco | 99631 | fperczyk@fi.uba.ar |
| Sobico Carla | 99738 | csobico@fi.uba.ar |

Entregas realizadas:



Índice

| | | |
|----------|--|-----------|
| 1 | Objetivos | 2 |
| 2 | Desarrollo | 2 |
| 2.1 | Diseño | 2 |
| 2.2 | Clases, estructuras y funciones | 3 |
| 2.3 | Pruebas | 3 |
| 2.3.1 | Desempeño | 4 |
| 2.3.2 | Validez del formato de las transacciones | 6 |
| 3 | Conclusiones | 9 |
| 4 | Referencias | 9 |
| 5 | Códigos fuente | 10 |
| 5.1 | Códigos provistos por la cátedra | 30 |
| 6 | Enunciado | 38 |

1 Objetivos

El presente trabajo tiene como objetivo la construcción de un bloque a partir del procesamiento de transacciones. Este bloque es parte de la *Algochain* (simplificación de la tecnología de la *blockchain*), que en conjunto con bloques forman una cadena, generando el sistema de la criptomoneda simulada.

2 Desarrollo

Para el desarrollo del objetivo propuesto se utiliza el lenguaje de programación C++. En la sección 2.1 se presentan las tareas a realizar y el diseño de las soluciones, en la sección 2.2 las clases, estructuras y funciones a utilizar para la implementación, en la sección 2.3 las corridas de pruebas y por último en la sección 5 el programa final.

2.1 Diseño

Para diseñar este trabajo práctico Se hizo uso de las estrategias *bottom-up* y *top-down* haciendo especial énfasis en la primera, puesto que se prefirió diseñar a partir de los niveles más bajos pero sin perder de vista el objetivo de cada clase/estructura y el alcance que debían tener.

Se debe tener en cuenta que toda transacción de *Algocoin* (criptomoneda simulada en el presente trabajo práctico) pertenece a un bloque, tal como en el sistema de *Bitcoin*s. Estos bloques están formados por un *header* y un *body*. Dentro del *header* se encuentra el *hash* del bloque antecesor en la *blockchain* (*prev_block*), el *hash* de todas las transacciones incluidas en el bloque (*txns_hash*), la dificultad con la cual fue ensamblado el bloque (*bits*) y, por último, un campo arbitrario para alterar el *hash* resultante de todo el *header* (*nonce*). Mientras que el cuerpo del bloque contiene la cantidad total de transacciones (*txn_count*) y una secuencia de dichas transacciones (*txns*). Estas transacciones están compuestas por el número de transacciones de entrada (*n_tx_in*), una secuencia con las entradas de donde se recibieron las *bitcoins* (*inputs*), el número de transacciones de salida (*n_tx_out*) y una secuencia de transacciones de salida (*outputs*). El *input* está compuesto por un *outpoint*, conformado por un indicador del *hash* de la transacción previa (*tx_id*) y un índice dentro de la secuencia de *outputs* de dicha transacción de entrada (*idx*), y por el *hash* criptográfico de la clave pública del destinatario (*addr*). El *output* está compuesto un valor que indica la cantidad de *bitcoins* a transferir (*value*) y el *hash* criptográfico de la clave pública del destinatario (*addr*).

Realizar un *hash* se refiere a aplicar la función de *hash* SHA256(), esta función utilizada para encriptar que devuelve una salida de 32 bytes representada con 64 dígitos hexadecimales.

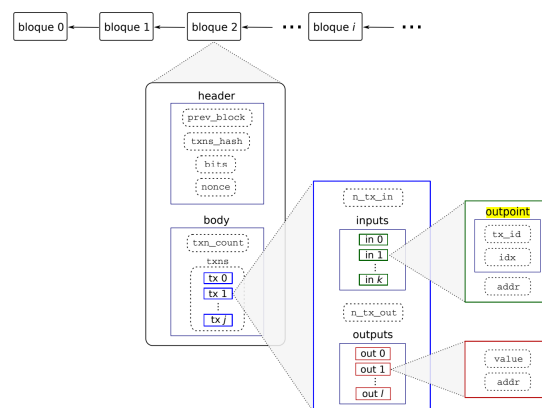


Figura 2.1: Esquema de alto nivel de la *Algochain*

Aclarada la estructura base del problema a resolver e implementar, se decidió que los niveles superiores dentro de la jerarquía debían ser clases: los bloques, *header* y *body*, transacciones (txns), *output* e *input*; pues representan a objetos abstractos con características únicas al momento de analizar las transacciones y la *Algochain* en sí. Dado que los *outpoints* están relacionados únicamente con los *inputs*, sólo éstos pueden modificar sus valores, por lo que se optó por dejarlo como una estructura.

En algunas de estas clases fue necesaria la implementación vectores para almacenar diferentes tipos de datos, por lo que se implementaron los mencionados arreglos como una clase y un *template*. Para la dimensión del vector de transacciones se usó una estrategia exponencial, es decir se incrementa la memoria destinada para un arreglo multiplicando la pedida previamente por una constante.

Luego se llegó al punto donde se debió elegir como estarán organizadas las secuencias de transacciones y las transacciones de entrada y salida. Se prefirió utilizar un *template* de arreglo para almacenar estas secuencias, ya que hace al código más fácilmente escalable y su uso es intuitivo.

2.2 Clases, estructuras y funciones

Se diseñó pensando que cada clase debería autocontenerse. Es decir: ni clases jerárquicamente superiores ni inferiores pueden acceder a sus atributos; sus valores se modifican y obtienen a partir de *getters* y *setters*. Así mismo, si se implementara el uso del concepto de herencia (relación de clases padres-hijos), se podría cambiar el alcance de las clases superiores por sobre las inferiores y dejaría de existir la autocontención mencionada.

Se decidió que cada clase debía contener un método del tipo *getAsString*, ya que son necesarios para poder imprimir un bloque entero en el flujo de salida y realizar el *hash* del vector de transacciones.

El programa está compuesto por un *main.cpp*, *cmdline.h*, *sha256* y *block.h* donde se definen las variables globales para poder manejar el flujo de entrada y salida y sus funciones de lectura. Además contiene un archivo *tools.h* con funciones auxiliares necesarias para algunos procedimientos, tales como: verificar si el atributo de un objeto es un número o un hash; o la conversión de hexadecimal a binario para el procesamiento de la dificultad especificada.

2.3 Pruebas

Se diseñaron pruebas para verificar la robustez y desempeño del algoritmo implementado para la carga e impresión de un bloque. Se pueden categorizar en 2 subgrupos orientados a probar distintos aspectos del código: desempeño y verificación de formato.

Para compilar este programa se realizó el siguiente *Makefile*:

```
CCFLAGS= -Wall -pedantic
CC= g++

all: algochain

algochain: main.o cmdline.o sha256.o
$(CC) $(CCFLAGS) -o algochain main.o cmdline.o sha256.o

main.o: main.cc cmdline.h sha256.h Array.h block.h main.h
$(CC) $(CCFLAGS) -c main.cc -o main.o

cmdline.o: cmdline.cc cmdline.h
$(CC) $(CCFLAGS) -c cmdline.cc -o cmdline.o

sha256.o: sha256.cpp sha256.h
```

```
$(CC) $(CCFLAGS) -c sha256.cpp -o sha256.o
```

```
clean:
```

```
$(RM) *.o cmdline
```

Luego para correrlo por consola se escribe la siguiente linea

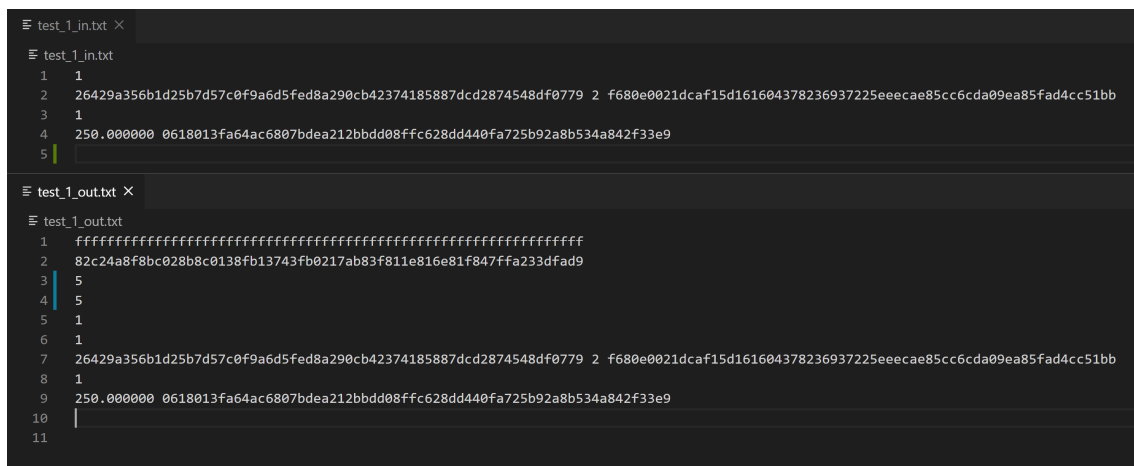
```
./algochain -i <archivo de entrada> -o <archivo de salida> -d <nivel de dificultad>
```

2.3.1 Desempeño

Se probó el desempeño del código en cuanto al rendimiento de las entradas, el procesamiento de las mismas y su posterior impresión incrementando la cantidad de entradas: entrada unitaria, cien mil entradas, un millón de entradas.

Si bien la dificultad que se puede llegar a tener en el sistema es 256, sólo se comprobó el rendimiento del código ante múltiples dificultades hasta tener tiempos de espera para la finalización de la corrida superiores al minuto.

En una primera instancia se probó ingresar una entrada con una única transacción, de una entrada y una salida con un formato correcto (asumiendo que los datos de la transacción son correctos). El resultado de esta prueba se muestran en la figura



```
test_1_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
5

test_1_out.txt
1 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2 82c24a8f8bc028b8c0138fb13743fb0217ab83f811e816e81f847ffa233dfad9
3 5
4 5
5 1
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
8 1
9 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
10
11
```

Figura 2.2: Entrada y salida para una transacción de dificultad 5.

Luego se probó ingresar una mayor cantidad de transacciones válidas. Se fueron aumentando progresivamente la cantidad de transacciones hasta llegar a 1 millón. El resultado se muestra en la figura



```

test_2_in.txt X
test_2_in.txt
999994 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
999995 1
999996 250.5 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
999997 1
999998 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
999999 1
1000000 250.5 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
1000001

test_2_out.txt X
test_2_out.txt
1000000 1
1000001 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
1000002 1
1000003 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
1000004 1
1000005 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
1000006
1000007

test_2_out.txt X
test_2_out.txt
1 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2 32dab873540fc44554f97cf9ba1c882a6d31151d0ed239484d534a470cb27106
3 5
4 42
5 250000
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
8 1

```

Figura 2.3: Entrada y salida para una transacción de dificultad 5.

Se realizó la prueba para el nivel de dificultad, por el algoritmo implementado esta dificultad se podría buscar hasta 256 bits. Se decidió buscar para que dificultad se lograba una velocidad menor a 1 minuto. Para nivel de dificultad 25 (Fig. 2.4) se encontró que tardaba en finalizar el programa 18 minutos, mientras que para dificultad 20 (Fig. 2.5) finaliza en 30 segundos.

```

test_1_in.txt X
test_1_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
5

test_1_out.txt X
test_1_out.txt
1 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2 82c24a8f8bc028b8c0138fb13743fb0217ab83f811e816e81f847ffa233dfad9
3 25
4 40583121
5 1
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
8 1
9 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
10
11

```

Figura 2.4: Entrada y salida para una transacción de dificultad 25.

El *hash* doble del *header* queda:

00000000cdcbc7f05359565bd8daa941dce8ad7fcc3532193bf5bda1264521cc6

```

test_1_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
5

test_1_out.txt
1 ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2 82c24a8f8bc028b8c0138fb13743fb0217ab83f811e816e81f847ffa233dfad9
3 20
4 409341
5 1
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
8 1
9 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
10
11

```

Figura 2.5: Entrada y salida para una transacción de dificultad 20.

El *hash* doble del *header* queda:

000004e63ab1d8a08cbece4b261391cecce899890766350e5fdc10fd192c8c0f

2.3.2 Validez del formato de las transacciones

Dado que el formato que deben tener las transacciones de entrada debe respetarse de manera estricta, se realizaron pruebas consistentes en ingresar entradas con pequeñas variaciones cada una: espacios o comienzos de línea de más; incongruencia entre el número que representa la cantidad de entradas a recibir y la cantidad de entradas real, y su análogo para las salidas; y caracteres que no corresponden o sin sentido.

La detección de errores se implemento en los métodos de seteo de cada objeto, es decir cada objeto se ocupa de cargar y validar los datos. En esta primera instancia se **trabajo** informando a niveles superiores y abortando la ejecución del programa, previamente informando en que transacción se produce el error. La forma de informar es seteando un atributo de este objeto con un texto en particular, si bien esto funciona correctamente en un futuro se **implementara** con excepciones.

```

Codigos > test_4_in.txt
1 2
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
5 1
6 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
7 1
8 250.000000 0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9

francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_4_in.txt -o test_4_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.6: Entrada con un error en el ntxin.

Se puede notar que ntxin es 2 pero hay solo un input. El programa reconoce y muestra por consola donde se produce el error.

```

Codigos > test_5_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
3 2
4 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
5 1
6 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
7 1
8 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_5_in.txt -o test_5_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.7: Entrada con un error en el ntxout.

```

Codigos > test_6_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
3 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
4 1
5 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
8 1
9 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_6_in.txt -o test_6_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.8: Entrada con un error en el ntxin.

```

Codigos > test_7_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
3 1
4 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
5 2
6 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
8 5
9 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
10 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
11 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
12 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
13 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

test_7_out.txt X
Codigos > test_7_out.txt
1 |ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
2 aefb8dcd6e33cbb7d563fa503921805a4f796d903a384c27d973115b82e89824
3 3
4 14
5 2
6 1
7 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
8 1
9 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
10 2
11 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
12 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51b1
13 5
14 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
15 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
16 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
17 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
18 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

```

Figura 2.9: Entrada y salida con dos transacciones de diferentes cantidad de inputs y outputs.


```

Codigos > test_8_in.txt
1 1
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
5 ?
6 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
7 1
8 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: bash +
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_8_in.txt -o test_8_out.txt -d 3
Error en la transaccion 2
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.10: Entrada con un caracter invalido en la transacción 2.

```

Codigos > test_9_in.txt
1 2
2
3
4
5 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
6 1
7 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
8 1
9 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
10 1
11 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: bash +
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_9_in.txt -o test_9_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.11: Entrada con un error de formato, enters donde no se debe.

```

Codigos > test_10_in.txt
1 2
2 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
3 1
4 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9
5 1
6
7
8
9 26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779 2 f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
10 1
11 250.000000 0618013fa64ac6807bdea212bdd08ffc628dd440fa725b92a8b534a842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: bash +
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_10_in.txt -o test_10_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque

```

Figura 2.12: Entrada con un error de formato, enters donde no se debe.

```

Codigos > test_11_in.txt
1 1
2 48df0779 2 d4cc51bb
3 1
4 250.5 842f33e9

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: bash +
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$ ./algochain -i test_11_in.txt -o test_11_out.txt -d 3
Error en la transaccion 1
Carga de datos interrumpida
Vuelva a cargar los datos del bloque
francisco@piki:~/Desktop/Facultad/Algoritmos II/TP0/Codigos$

```

Figura 2.13: Entrada con los largos de los hash.

3 Conclusiones

Se pudo cumplir con el objetivo del trabajo práctico: realizar la carga efectiva de transacciones dada por un flujo de entrada y su posterior impresión por un flujo de salida.

Si bien el código fue probado bajo distintos criterios de funcionamiento (como se detalla en la sección 2.3) se vió que funciona correctamente para una amplia cantidad de transacciones de entrada y salida, lo que permite tener bloques con capacidad virtualmente infinita.

Además se verificó que responde bien a distintos tipos de formatos inválidos de las transacciones provenientes del flujo de entrada, indicando el número de transacción donde ocurrió el error, permitiendo así el rastreo del error y, por lo tanto, un código confiable y robusto.

4 Referencias

- Hash SHA256
<https://emn178.github.io/online-tools/sha256.html>
- Programación en C++
Paul Deitel, Harvey Deitel - C++ How to Program, 8th Edition-Prentice Hall (2011)
- Fundamentos de la ingeniería de software
Ghezzi, C., Jazayeri, M., Mandrioli, D., Fundamentals of Software Engineering, Prentice-Hall International, Singapore, 1991

5 Códigos fuente

Listing 1: Array.h

```

1  #ifndef ARRAY_INCLUDED
2  #define ARRAY_INCLUDED
3  #include <iostream>
4  using namespace std;

5  template<class T>
6  class Array
7  {
8  public:
9      Array(); //Creador base
10     Array( const Array<T> & ); //Creador-Copiador
11     Array(size_t); //Creador mediante tamaño del array
12     ~Array(); //Destructor
13     void ArrayRedim(size_t); //Redimensionador de arrays.
14     size_t getSize(); //Método: determina el tamaño del array
15     Array<T> &operator=( const Array<T> & ); //Operador asignación para una
16     array: A=B, donde A y B son arrays. Recibe como parámetro un array por
17     referencia constante, para no modificar lo que tiene dentro
18     bool operator==( const Array<T> & ); //Operador lógico para comprobar si
19     son iguales 2 arrays. Recibe como parámetro un array por referencia
20     constante, para no modificar lo que tiene dentro
21     T &operator[](size_t); //Operador indexación: Retorna un elemento del
22     vector (se puede cambiar, pues se retorna por referencia)
23     template <class TT>
24     friend std::ostream &operator<<(std::ostream&, Array <TT> &); //Operador de
25     impresion de salida
26     // void mergeSort(); //Implementacion de metodo de ordenamiento MERGE SORT
27     // void printBackwards_recursive(); //Impresion del arreglo del ultimo al
28     primero
29     // void printForwards_recursive(); //Impresion del arreglo del primero al
30     ultimo
31     // double sumOfElements_recursive(); //Calcula la suma de los elementos del
32     vector
33     bool empty(); // Verifica si un arreglo esta vacio.

34 private:
35     size_t rsize; //Atributo que indica el tamaño del array
36     size_t vsize;
37     T *ptr; //Atributo que indica la dirección donde inicia el puntero

38     // void _mergeSort(int, int);
39     // void _merge(int,int,int);
40     // void _printBackwards(int);
41     // void _printForwards(int);
42     // double _sumOfElements(int);
43 };

44 template <class T>
45 Array<T>::Array()
46 {
47     ptr = new T[10];
48     rsize = 10;
49     vsize = 10;
50 }

51 template <class T>
52 Array<T>::Array(const size_t init_size)
53 {
54     ptr = new T[init_size];
55     rsize = init_size;
56     vsize = init_size;
57 }

```

```
56
57
58 template <class T>
59 Array<T>::Array(const Array<T> &init)
60 {
61     rsize = init.vsize;
62     vsize = init.vsize;
63
64     ptr = new T[init.vsize];
65     for(size_t i=0; i< init.vsize; i++)
66     {
67         ptr[i] = init.ptr[i]; //ASUMO QUE T tiene el operador =
68     }
69 }
70
71
72 template <class T>
73 Array<T>::~~Array() //Implementación del destructor de Array
74 {
75     if (ptr)
76         delete [ ] ptr; //Destructor de memoria dinámica.
77 }
78
79
80 template <class T>
81 size_t Array<T>::getSize() { return vsize; } //Implementación del getter del
82     tamaño del Array.
83
84
85 template <class T>
86 Array<T>& Array<T>::operator=(const Array<T> &right)
87 {
88     if(&right != this)
89     {
90         if(rsize != right.vsize)
91         {
92             T *aux;
93             aux = new T[right.vsize];
94             delete [ ] ptr;
95             rsize = right.vsize;
96             ptr = aux;
97
98             for (size_t i = 0; i<right.vsize; i++)
99             {
100                 ptr[i] = right.ptr[i];
101             }
102             vsize = right.vsize;
103             return *this;
104         }
105         return *this;
106     }
107
108 template <class T>
109 bool Array<T>::operator==(const Array<T> &right)
110 {
111     if(vsize != right.vsize)
112     {
113         return false;
114     }
115     else
116     {
117         for(int i = 0; i<right.vsize; i++)
118         {
119             if(ptr[i] != right.ptr[i])
120             {
121                 return false;
122             }
123         }
124         return true;
125     }
126 }
```

```
126 }
127
128 template<class T>
129 T &Array<T>::operator[](size_t subscript)
130 {
131     if(subscript >= vsize)
132         std::abort();
133     return ptr[subscript];
134 }
135
136 template <class T>
137 std::ostream &operator <<(std::ostream &os, Array<T> &arr)
138 {
139     int i, size;
140     size = arr.getSize();
141     for(i=0; i<size; i++)
142     {
143         os << arr[i] << endl; //asumo que T tiene sobrecargado el <<
144     }
145     return os;
146 }
147
148 template<class T>
149 void Array<T>::ArrayRedim(size_t new_size)
150 {
151     if (new_size > rsize) {
152         T *aux = new T[new_size];
153         for (size_t i = 0; i < vsize; ++i)
154             aux[i] = ptr[i];
155         delete[] ptr, ptr = aux;
156         rsize = new_size;
157     }
158
159     vsize = new_size;
160 }
161
162 template <class T>
163 bool Array<T>::empty()
164 {
165     if(vsize)
166         return true;
167     else
168         return false;
169 }
170
171 #endif //ARRAY_INCLUDED
```

Listing 2: Block.h

```

1 #include <iostream>
2 #include <string.h>
3 #include <cstring>
4 #include "Array.h"
5 #include "math.h"
6 #include <fstream>
7 #include <sstream>
8 #include <cstdlib>
9 #include <bitset>
10 #include <ctype.h>
11 #include "tools.h"
12
13 // o Se usaran strings para representar los hash
14 //
15 // o Jerarquía de clases: Bloque > Header/Body>Txn > Input/Output > Outpoint (
16 //   estructura que pertenece a Input).
17
18 //-----ESTRUCTURA OUTPOINT
19 -----
20 struct outpnt
21 {
22     string tx_id; //Es un hash de la transaccion de donde este input toma fondos
23     size_t idx; //Valor entero no negativo que representa un indice sobre la
24     //   secuencia de outputs de la transaccion con hash tx id
25 };
26 //-----CLASE INPUT
27 -----
28 class inpt
29 {
30     outpnt outpoint;
31     string addr; //La direccion de origen de los fondos (que debe coincidir con la
32     //   direccion del output referenciado)
33     public:
34     inpt(); //Creador base
35     inpt(string&); //Creador mediante una string
36     ~inpt(); //Destructor
37     //Si hay getters deberian haber setters. Si no se usan, eliminarlos.
38     string getAddr();
39     outpnt getOutPoint();
40     string getInputAsString();
41 };
42 inpt::inpt(){} //Creador base
43
44 inpt::~inpt(){} //Destructor base
45
46 inpt::inpt(string & str) //Creador mediante una string
47 {
48     istringstream ss(str);
49     string str_tx_id, str_id, str_addr;
50
51     getline(ss, str_tx_id, ' '); // Se recorren los campos. Si el formato es
52     //   erroneo, se detecta como una cadena vacia.
53     getline(ss, str_id, ' ');
54     getline(ss, str_addr, '\n');
55
56     if((isHash(str_tx_id)==true) && (isNumber<size_t>(str_id)==1) && (isHash(
57     str_addr)==true))
58     {
59         this->outpoint.tx_id=str_tx_id;
60         this->outpoint.idx=stoi(str_id);
61         this->addr=str_addr;
62     }
63     else
64     {
65         this->addr=ERROR; // Si hay un error pone addr en ERROR, para avisar a un

```

```

        nivel mas alto
    }
}

outpt inpt::getOutPoint(){return outpoint;}

string inpt::getAddr(){return addr;}

string inpt::getInputAsString()
{
    stringstream ss;
    string aux;

    ss<<(this->getOutPoint().idx);    //Pasaje de size_t
    ss>>aux;                          // a string

    string result;
    result.append((this->getOutPoint()).tx_id);
    result.append(" ");
    result.append(aux);
    result.append(" ");
    result.append((this->getAddr()));

    return result;
}
//-----CLASE OUTPUT
-----

class outpt
{
    double value; //La cantidad de Algocoins a transferir en este output
    string addr; //La direccion de origen de los fondos (que debe coincidir con la
                direccion del output referenciado)

public:

    outpt(); //Creador base
    outpt(string&); //Creador mediante una string
    ~outpt(); //Destructor
    double getValue();
    string getAddr();
    string getOutputAsString();
};

outpt::outpt() //Creador base
{
}

outpt::~~outpt() //Destructor base
{
}

outpt::outpt(string & str) //Creador mediante una string
{
    istringstream ss(str);
    string str_value, str_addr;

    getline(ss, str_value, ' '); // Se recorren los campos. Si el formato es
    errneo, se detecta como una cadena vacía.
    getline(ss, str_addr, '\n');

    if((isNumber<double>(str_value)==1) && (isHash(str_addr)==true))
    {
        this->value=stoi(str_value);
        this->addr=str_addr;
    }
    else
    {
        this->addr=ERROR;
    }
}

```

```

    }
126
    string outpt::getAddr(){return addr;}
128
    double outpt::getValue(){return value;}
130
    string outpt::getOutputAsString()
132 {
        string aux;
134         string result;
        aux=to_string(this->getValue());
136         result.append(aux);
        result.append(" ");
138         result.append((this->getAddr()));
        return result;
140     }

142 //-----CLASE TXN
    -----

144
    class txn
146 {
        private:
148         size_t n_tx_in; //Indica la cantidad total de inputs en la transaccion
        size_t n_tx_out; //Indica la cantidad total de outputs en la transaccion
150         Array<inpt> tx_in; //Datos de entrada para las transacciones
        Array<outpt> tx_out; //Datos de salida para las transacciones
152
        public:
154         txn(); //Creador base
        ~txn(); //Destructor
156
        void setNTxIn(const size_t) ;
158         void setNTxOut(const size_t);
        bool setTxIn(const size_t n, istream *iss); // Seteador que valida los datos y
            devuelve un booleano para el error
160         bool setTxOut(const size_t n, istream *iss);

162         size_t getNTxIn();
        size_t getNTxOut();
164         Array<inpt>& getInputs();
        Array<outpt>& getOutPuts();
166
        string getTxnAsString();
168         string validateTxn();
    };

170

172 txn::txn()
    {
174         n_tx_in=0;
        n_tx_out=0;
176         tx_in.ArrayRedim(0);
        tx_out.ArrayRedim(0);
178     }

180 txn::~txn()
    {
182     }

184 void txn::setNTxIn(const size_t n)
    {
186         n_tx_in=n;
        if(tx_in.getSize() == 0)
188         {
            tx_in.ArrayRedim(n);
190         }
    }

```



```

192
194 void txn::setNTxOut(const size_t n)
195 {
196     n_tx_out=n;
197     if(tx_out.getSize() == 0)
198     {
199         tx_out.ArrayRedim(n);
200     }
201 }
202
204 bool txn::setTxIn(const size_t n, istream *iss) //Se modifica el retorno del
        setter por defecto (void) por
        // necesidad. Verifica si el setteo pudo realizarse
        correctamente.
206 string aux_s;
207 for (size_t i = 0; i < n; i++)
208 {
209     getline(*iss, aux_s, '\n');
210     inpt in(aux_s);
211     if(isError(in.getAddr())==false)
212         return false;
213     tx_in[i] = in;
214 }
215 return true;
216 }
218
219 bool txn::setTxOut(const size_t n, istream *iss) //Se modifica el retorno del
        setter por defecto (void) por
        // necesidad. Verifica si el setteo pudo realizarse
        correctamente.
222 string aux_s;
223 for (size_t i = 0; i < n; i++)
224 {
225     getline(*iss, aux_s, '\n');
226     outpt out(aux_s);
227     if(isError(out.getAddr())==false)
228         return false;
229     tx_out[i] = out;
230 }
231 return true;
232 }
234 size_t txn::getNTxIn(){return n_tx_in;}
236 size_t txn::getNTxOut(){return n_tx_out;}
238 Array<inpt>& txn::getInputs(){return tx_in;}
239 Array<outpt>& txn::getOutPuts(){return tx_out;}
240
241 string txn::getTxnAsString()
242 {
243     string result, aux;
244     aux = to_string(n_tx_in);
245     result.append(aux);
246     result.append("\n");
247     for(size_t i = 0; i < n_tx_in; i++)
248     {
249         result.append(tx_in[i].getInputAsString());
250         result.append("\n");
251     }
252     aux = to_string(n_tx_out);
253     result.append(aux);
254     result.append("\n");
255     for(size_t i = 0; i < n_tx_out; i++)
256     {
257         result.append(tx_out[i].getOutputAsString());

```

```

258     result.append("\n");
260     return result;
262 }
262 //-----CLASE BODY
-----
264 class bdy
265 {
266     size_t txn_count;
267     Array <txn> txns;
268 public:
269     bdy();
270     ~bdy();
271     bdy getBody();
272     string getBodyAsString();
273     size_t getTxnCount();
274     Array<txn> getTxns();
275     string getTxnAsString();
276     string setTxns(istream *iss);
277     void setTxnCount(const size_t n);
278     void txnsArrRedim(const size_t );
279 };
280 bdy::bdy()
281 {
282 }
284 bdy::~bdy(){}
286 void bdy::setTxnCount(const size_t n)
287 {
288     txn_count = n;
289     if(this->txns.getSize() == 0)
290     {
291         txns.ArrayRedim(n);
292     }
293 }
294 string bdy::setTxns(istream *iss)
295 {
296     string str,error_string;
297     size_t aux, i = 0;
298     bool err;
300     while(getline(*iss, str, '\n'))
301     {
302         if(i >=txns.getSize())
303         {
304             txns.ArrayRedim(txns.getSize()*2); // Dependiendo de cuantos datos haya
305             // que analizar se puede modificar
306             // la estrategia de crecimiento del arreglo.
308             // Se verifica n_tx_in
309             if(isNumber<size_t>(str)==0 || (str[0]) == '\0')
310             {
311                 err=true;
312                 break;
313             }
314             aux = stoi(str);
315             txns[i].setNTxIn(aux);
318             // Se verifican las entradas
319             if(txns[i].setTxIn(aux, iss)==false)
320             {
321                 err=true;
322                 break;
323             }
324 }

```

```

326 // Se verifica n_tx_out
getline(*iss, str, '\n');

328 if(isNumber<size_t>(str)==0 || (str[0] == '\0'))
{
330     err=true;
    break;
332 }
    aux = stoi(str);
334 txns[i].setNTxOut(aux);

336 // Se verifican las salidas
if(txns[i].setTxOut(aux, iss)==false)
338 {
    err=true;
340     break;
    }
342 i++;
}
344 if(err==true)
{
346     error_string.append("Error en la transaccion ");
    error_string.append(to_string(i+1));
348     error_string.append("\n");
    error_string.append("Carga de datos interrumpida");
350     error_string.append("\n");
    error_string.append("Vuelva a cargar los datos del bloque");
352     return error_string;
}
354 txn_count = i;
return "\0";
356 }

358 string bdy::getBodyAsString()
{
360     string result, str;
    str = to_string(txn_count);
362     result.append(str);
    result.append("\n");
364
    for (size_t i = 0; i < txn_count; i++)
366     {
        result.append(txns[i].getTxnAsString());
368     }
    return result;
370 }

372 size_t bdy::getTxnCount(){return txn_count;}
Array<txn> bdy::getTxns(){return txns;}
374
bdy bdy::getBody(){return *this;}
376
void bdy::txnsArrRedim(const size_t n ){txns.ArrayRedim(n);}
378
//-----CLASE HEADER
-----

380
class hdr
382 {
    string prev_block; //El hash del bloque completo que antecede al bloque actual
                        // en la Algochain.
384     string txns_hash; //El hash de todas las transacciones incluidas en el bloque.
    size_t bits;      // Valor entero positivo que indica la dificultad con la que
                        // fue minada este bloque.
386     size_t nonce;    // Un valor entero no negativo que puede contener valores
                        // arbitrarios. El objetivo de este
                        // campo es tener un espacio de prueba modificable para poder generar
                        // hashes sucesivos hasta
388     // satisfacer la dificultad del minado.

```

```

public:
390 hdr();
    ~hdr();
392 bool setPrevBlock(const string&);
    void setTxnsHash(const string&);
394 void setBits(const size_t n);
    void setNonce(const string prev_block, const string txns ,const size_t bits);
396 string getPrevBlock();
    string getTxnHash();
398 size_t getBits();
    size_t getNonce();
400 string getHeaderAsString();
};

402
404 hdr::hdr()
{
    prev_block = "\0";
406 txns_hash = "\0";
    bits = 0; //Se podría hacer que los bits y el nonce fueran ints para
              //detectar errores haciendo que estos valgan -1 (por ej)
408 nonce = 0;
}

410
412 hdr::~hdr(){}

414 bool hdr::setPrevBlock(const string & str)//Se modifica el retorno del setter
    por defecto (void) por
    // necesidad. Verifica si el setteo pudo realizarse
    correctamente.
{
    if(isHash(str) == false)
418 return false;
    else
420 {
        prev_block = str;
422 return true;
    }
424 }

426 void hdr::setTxnsHash(const string & str){txns_hash = sha256(sha256(str));}

428
430 void hdr::setBits(const size_t n){bits = n;}

432
434 string hdr::getPrevBlock(){return prev_block;}

436 string hdr::getTxnHash(){return txns_hash;}

438
440 size_t hdr::getBits(){return bits;}

442 size_t hdr::getNonce(){return nonce;}

444
446 string hdr::getHeaderAsString()
{
    string str;
448 string bit_string = to_string(bits); //convierto el bits a string y lo agrego
    a la string
    string nonce_string; //para guardar cuando transforme en string del nonce
450
    str.append(prev_block);//pongo primero en la string el prev block
452 str.append("\n");//PREGUNTAR si no viene con el barra n, creo que no pero si
    no ya esta
    str.append(txns_hash);//agrego el txns, ver comentario de la línea 25

```

```

454     str.append("\n");
455     str.append(bit_string);
456     str.append("\n");
457     nonce_string = to_string(nonce); //convierto el nonce a string
458     str.append(nonce_string);
459     return str;
460 }
461 void hdr::setNonce(const string prev_block, const string txns, const size_t bits) // Setea el header con el nonce que verifica que el hash del header
    cumpla con los primeros d bits en cero
462 {
463     size_t out = 0; //inicializo d_aux que contara el nivel de dificultad y out
    que es un flag para el for
464     size_t nonce_aux = 0; //inicializo el nonce, para mi hay que hacerlo double
    porque se puede hacer muy grande pero hay que cambiar struct
465
466     string header_str; //defino un header_aux auxiliar para hacer la string antes
    de hashearla
467     string hash_header; // para guardar el hash del header_aux
468
469     int j, i, aux;
470
471     int cant_char = bits/4; // me da la cantidad de char en 0 que necesito
472     int cant_bit = bits%4; //me da la cantidad de bits del ultimo char en 0
473
474     for (nonce_aux = 0; out == 0; nonce_aux++) //aumento el nonce hasta que el flag
    out sea 1, igualmente tambien hay un break, es por las dudas que el break
    no funcione como espero
475     {
476         header_str.clear();
477         nonce = nonce_aux;
478         header_str = getHeaderAsString();
479         header_str.append("\n");
480         hash_header = sha256(sha256(header_str)); //calculo el hash del header_aux
481         i=0;
482         aux=0;
483         while (i<cant_char)
484         {
485             if(hash_header[i] != '0')
486             {
487                 aux=1;
488                 break;
489             }
490             i++;
491         }
492         if(aux==1)
493         {
494             continue;
495         }
496         else
497         {
498             hash_header = Hex2Bin(hash_header.substr(i,i+1));
499
500             j=0;
501             aux=0;
502             while (j<cant_bit)
503             {
504                 if(hash_header[j] != '0')
505                 {
506                     aux=1;
507                     break;
508                 }
509                 j++;
510             }
511             if(aux == 0)
512             {
513                 nonce = nonce_aux; //guardo el nonce en el header_aux
514                 out = 1; // Para verificar que termine el for

```

```

        break; //como cumple la cantidad de bits necesarias y ya esta guardado y
        hasheado salgo
518     }
519 }
520 }
521 return ;
522 }

524 //-----CLASE BLOCK
    -----

class block
526 {
    private:
528     hdr header;
    bdy body;
530
    public:
532     block(); //Creador base
    block(const string,const size_t, istream*); //Creador en base al hash del
        bloque previo, al nivel de
534         //dificultad y un flujo de entrada por el que se reciben
            las
                // transacciones.
536     ~block(); //Destructor
    void setHeader(const string&,const size_t);
538     void setBody(istream *iss);

540     string getBlockAsString();
};

542 void block::setHeader(const string& prev_block_str,const size_t diffic)
544 {
    string aux;
546     header.setPrevBlock(prev_block_str);
    aux = body.getBodyAsString();
548     header.setTxnsHash(aux);
    header.setBits(diffic);
550     header.setNonce(header.getPrevBlock(),header.getTxnHash(),header.getBits());
}

552 void block::setBody(istream *iss)
554 {
    string str;
556     body.setTxnCount(0);
    body.txnsArrRedim(1); //Se inicializa en uno. Tiene redimensionamiento
        automatico a
558         // traves de metodos de la clase.
    if((str=body.setTxns(iss))!="\0")
560     {
        cerr<<str<<endl;
562         exit(1);
    };
564 }

566 block::block(const string str,const size_t diffic, istream *iss)
{
568     setBody(iss);
    setHeader(str, diffic);
570 }

572 block::~~block()
{
574 }

576 string block::getBlockAsString()
{
578     string result, str;
    result.append(header.getHeaderAsString());
580     result.append("\n");
}

```

```
582 |         result.append(body.getBodyAsString());  
    |         return result;  
    |     }
```

Listing 3: Main.cc

```
1  #include <fstream>
2  #include <iomanip>
3  #include <iostream>
4  #include <sstream>
5  #include <cstdlib>
6
7  #include "cmdline.h"
8  #include "sha256.h"
9  #include "block.h"
10 #include "main.h"
11
12 using namespace std;
13
14 int
15 main(int argc, char * const argv[])
16 {
17     string str = "fffffffffffffffffffffffffffffffffffffffffffffffffffff
18 ";
19     cmdline cmdl(options); // Objeto con parametro tipo option_t (struct)
20                             // declarado globalmente. Ver línea 51 main.cc
21     cmdl.parse(argc, argv); // Metodo de parseo de la clase cmdline
22
23     if (iss->bad()) {
24         cerr << "cannot read from input stream."
25             << endl;
26         exit(1);
27     }
28
29     block block0(str, difficulty, iss);
30
31     str = block0.getBlockAsString();
32
33     if (oss->bad()) {
34         cerr << "cannot write to output stream."
35             << endl;
36         exit(1);
37     }
38     *oss << str << endl;
39 }
```


Listing 4: Main.h

```

1  #ifndef MAIN_H
2  #define MAIN_H

3
4  #include <fstream>
5  #include <iomanip>
6  #include <iostream>
7  #include <sstream>
8  #include <cstdlib>
9  #include "cmdline.h"
10
11 using namespace std;
12
13 ***** Elementos globales *****
14 static void opt_input(string const &);
15 static void opt_output(string const &);
16 static void opt_factor(string const &);
17 static void opt_help(string const &);
18
19 // Tabla de opciones de línea de comando. El formato de la tabla
20 // consta de un elemento por cada opción a definir. A su vez, en
21 // cada entrada de la tabla tendremos:
22 //
23 // o La primera columna indica si la opción lleva (1) o no (0) un
24 // argumento adicional.
25 //
26 // o La segunda columna representa el nombre corto de la opción.
27 //
28 // o Similarmente, la tercera columna determina el nombre largo.
29 //
30 // o La cuarta columna contiene el valor por defecto a asignarle
31 // a esta opción en caso que no está explícitamente presente
32 // en la línea de comandos del programa. Si la opción no tiene
33 // argumento (primera columna nula), todo esto no tiene efecto.
34 //
35 // o La quinta columna apunta al método de parseo de la opción,
36 // cuyo prototipo debe ser siempre void (*m)(string const &arg);
37 //
38 // o La última columna sirve para especificar el comportamiento a
39 // adoptar en el momento de procesar esta opción: cuando la
40 // opción es obligatoria, deberá activarse OPT_MANDATORY.
41 //
42 // Además, la última entrada de la tabla debe contener todos sus
43 // elementos nulos, para indicar el final de la misma.
44 //
45 static option_t options[] = {
46     {1, "i", "input", "-", opt_input, OPT_DEFAULT},
47     {1, "o", "output", "-", opt_output, OPT_DEFAULT},
48     {1, "d", "difficulty", NULL, opt_factor, OPT_MANDATORY},
49     {0, "h", "help", NULL, opt_help, OPT_DEFAULT},
50     {0, },
51 };
52
53
54 static int difficulty;
55 static istream *iss = 0; // Input Stream (clase para manejo de los flujos de
56 entrada)
57 static ostream *oss = 0; // Output Stream (clase para manejo de los flujos de
58 salida)
59 static fstream ifs; // Input File Stream (derivada de la clase ifstream que
60 deriva de istream para el manejo de archivos)
61 static fstream ofs; // Output File Stream (derivada de la clase ofstream que
62 deriva de ostream para el manejo de archivos)
63
64 static void
65 opt_input(string const &arg)
66 {
67     // Si el nombre del archivos es "-", usaremos la entrada

```

```

66 // estándar. De lo contrario, abrimos un archivo en modo
67 // de lectura.
68 //
69 if (arg == "-") {
70     iss = &cin; // Establezco la entrada estandar cin como flujo de entrada
71 }
72 else {
73     ifs.open(arg.c_str(), ios::in); // c_str(): Returns a pointer to an array
74     // that contains a null-terminated
75     // sequence of characters (i.e., a C-string) representing
76     // the current value of the string object.
77     iss = &ifs;
78 }
79 // Verificamos que el stream este OK.
80 //
81 if (!iss->good()) {
82     cerr << "cannot open "
83         << arg
84         << ". "
85         << endl;
86     exit(1);
87 }
88 }
89
90 static void
91 opt_output(string const &arg)
92 {
93     // Si el nombre del archivos es "-", usaremos la salida
94     // estándar. De lo contrario, abrimos un archivo en modo
95     // de escritura.
96     //
97     if (arg == "-") {
98         oss = &cout; // Establezco la salida estandar cout como flujo de salida
99     } else {
100         ofs.open(arg.c_str(), ios::out);
101         oss = &ofs;
102     }
103
104     // Verificamos que el stream este OK.
105     //
106     if (!oss->good()) {
107         cerr << "cannot open "
108             << arg
109             << ". "
110             << endl;
111         exit(1); // EXIT: Terminación del programa en su totalidad
112     }
113 }
114
115 static void
116 opt_factor(string const &arg)
117 {
118     istringstream iss(arg);
119
120     // Intentamos extraer el factor de la línea de comandos.
121     // Para detectar argumentos que únicamente consistan de
122     // números enteros, vamos a verificar que EOF llegue justo
123     // después de la lectura exitosa del escalar.
124     //
125     if (!(iss >> difficulty)
126         || !iss.eof()) {
127         cerr << "non-integer factor: "
128             << arg
129             << ". "
130             << endl;
131         exit(1);
132     }
133
134     if (iss.bad()) {

```

```
134         cerr << "cannot read integer factor."
           << endl;
136         exit(1);
138     }
139 }

140 static void
opt_help(string const &arg)
142 {
143     cout << "cmdline -f factor [-i file] [-o file]"
           << endl;
144     exit(0);
146 }

148 #endif //MAIN_H
```

Listing 5: Parse.cpp

```
1  #include <fstream>
2  #include <iomanip>
3  #include <iostream>
4  #include <sstream>
5  #include <cstdlib>
6
7
8  #include "cmdline.h"
9  #include "sha256.h"
10
11 using namespace std;
12
13 void setBlock(istream *iss);
14
15 void
16 setBlock(istream *iss)
17 {
18     int n_tx_in, n_tx_out, i, j = 0;
19     string aux_s;
20     Array <txns> txs;
21     while (*iss >> n_tx_in){
22         txs[j].setNTxIn(n_tx_in);
23         for(i = 0; i < n_tx_in; i++)
24         {
25             *iss >> aux_s;
26             inpt in(aux_s);
27             txs[j].setInpt(in);
28         }
29         *iss >> n_tx_out;
30         txs[j].setNTxOut(n_tx_out);
31         for(i = 0; i < n_tx_out; i++)
32         {
33             *iss >> aux_s;
34             outpt out(aux_s);
35             txs[j].setOutps(out);
36         }
37         j++;
38     }
39 }
40
41 void printBlock(*oss)
42 {
43     *oss << header->prev_block << endl;
44     *oss << header->txns.hash << endl;
45     *oss << header->bits << endl;
46     *oss << header->nonce << endl;
47
48     *oss << getTxNcount() << endl;
49
50     for para los
51         (body->txns)[j].printTransaccion(oss);
52 }
```



Listing 6: tools.h

```

1  #ifndef  TOOLS_H
2  #define  TOOLS_H

4  #include <iostream>
5  #include <string.h>
6  #include <cstring>
7  #include <sstream>
8  #include <cstdlib>
9  #include <bitset>

10
11  using namespace std;

12  static const string ERROR="Error"; //Se usa a modo de macro.

14  string Hex2Bin(const string& s); // Transforma una cadena de caracteres que
    contiene un numero en Hexa a una en binario
16  bool isHash(const string& str); // Confirma si str cumple con los requisitos
    minimos de un HASH
18  bool isNumber(const string& s); // Revisa si s es un numero, implementado como
    template para distinguir si es int o double, etc...
19  bool isError(const string& addr); // Se fija si en addr esta lo guardado en la
    variable ERROR

20
21  bool isError(const string& addr)
22  {
23      if(addr==ERROR)
24          return false;
25      else
26          return true;
27  }

28  string Hex2Bin(const string& s) //transforma de hexa a binario una string, maximo
    4 bytes, 8 char y 32 bits
29  {
30      stringstream ss;
31      ss << hex << s;
32      unsigned n;
33      ss >> n;
34      bitset<4> b(n); //32 es el maximo por el unsigned
35
36      return b.to_string(); //.substr(32 - 4*(s.length()));
37  }
38  bool isHash(const string& str) //Devuelve false (0) si no es un hash, true (1)
    si lo es.
39  {
40      int n;
41      if(str.length()!=64) //El hash devuelve una string de 64 chars. Es siempre de
        largo fijo.
42          return false;
43      for (size_t i = 0; i < str.length(); i++)
44      {
45          n=(int)(tolower(str[i])-'0');
46          if(n<0 || (n>9&& n<17) || (n>22 && n<49) || n>63) //se verifica si el char de la
            string es un numero hexa
47              return false;
48      }
49      return true;
50  }
51
52  template<typename Numeric>
53  bool isNumber(const string& s) //Devuelve 1 si es true y 0 si es false
54  {
55      Numeric n;
56      return((istringstream(s) >> n >> ws).eof());
57  }
58
59
60

```

```
|| #endif //TOOLS_H
```

5.1 Códigos provistos por la cátedra

Listing 7: cmdline.h

```
1  #ifndef _CMDLINE_H_INCLUDED_
2  #define _CMDLINE_H_INCLUDED_
3
4  #include <string>
5  #include <iostream>
6
7  #define OPT_DEFAULT    0
8  #define OPT_SEEN      1
9  #define OPT_MANDATORY 2
10
11 struct option_t {
12     int has_arg;
13     const char *short_name;
14     const char *long_name;
15     const char *def_value;
16     void (*parse)(std::string const &); // Puntero a función de opciones
17     int flags;
18 };
19
20 class cmdline {
21     // Este atributo apunta a la tabla que describe todas
22     // las opciones a procesar. Por el momento, sólo puede
23     // ser modificado mediante constructor, y debe finalizar
24     // con un elemento nulo.
25     //
26     option_t *option_table;
27
28     // El constructor por defecto cmdline::cmdline(), es
29     // privado, para evitar construir "parsers" (analizador
30     // sintáctico, recibe una palabra y lo interpreta en
31     // una acción dependiendo su significado para el programa)
32     // sin opciones. Es decir, objetos de esta clase sin opciones.
33     //
34
35     cmdline();
36     int do_long_opt(const char *, const char *);
37     int do_short_opt(const char *, const char *);
38 public:
39     cmdline(option_t *);
40     void parse(int, char * const []);
41 };
42
43 #endif
```

Listing 8: cmdline.cc

```

1 // cmdline - procesamiento de opciones en la línea de comando.
2 //
3 // $Date: 2012/09/14 13:08:33 $
4 //
5 #include <string>
6 #include <cstdlib>
7 #include <iostream>
8 #include "cmdline.h"
9
10 using namespace std;
11
12 cmdline::cmdline()
13 {
14 }
15
16 cmdline::cmdline(option_t *table) : option_table(table)
17 {
18     /*
19      * - Lo mismo que hacer:
20      *
21      * option_table = table;
22      *
23      * Siendo "option_table" un atributo de la clase cmdline
24      * y table un puntero a objeto o struct de "option_t".
25      *
26      * Se estaría contruyendo una instancia de la clase cmdline
27      * cargandole los datos que se hayan en table (la table con
28      * las opciones, ver el código en main.cc)
29      */
30 }
31
32 void
33 cmdline::parse(int argc, char * const argv[])
34 {
35     #define END_OF_OPTIONS(p) \
36         ((p)->short_name == 0 \
37          && (p)->long_name == 0 \
38          && (p)->parse == 0)
39
40     // Primer pasada por la secuencia de opciones: marcamos
41     // todas las opciones, como no procesadas. Ver código de
42     // abajo.
43     //
44     for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op)
45         op->flags &= ~OPT_SEEN;
46
47     // Recorremos el arreglo argv. En cada paso, vemos
48     // si se trata de una opción corta, o larga. Luego,
49     // llamamos a la función de parseo correspondiente.
50     //
51     for (int i = 1; i < argc; ++i) {
52         // Todos los parámetros de este programa deben
53         // pasarse en forma de opciones. Encontrar un
54         // parámetro no-opción es un error.
55         //
56         if (argv[i][0] != '-') {
57             cerr << "Invalid non-option argument: "
58                  << argv[i]
59                  << endl;
60             exit(1);
61         }
62     }
63
64     // Usamos "--" para marcar el fin de las
65     // opciones; todo los argumentos que puedan
66     // estar a continuación no son interpretados
67     // como opciones.
68     //

```



```

70     if (argv[i][1] == '-')
71         && argv[i][2] == 0)
72         break;
73
74     // Finalmente, vemos si se trata o no de una
75     // opción larga; y llamamos al método que se
76     // encarga de cada caso.
77     //
78     if (argv[i][1] == '-')
79         i += do_long_opt(&argv[i][2], argv[i + 1]);
80     else
81         i += do_short_opt(&argv[i][1], argv[i + 1]);
82 }
83
84 // Segunda pasada: procesamos aquellas opciones que,
85 // (1) no hayan figurado explícitamente en la línea
86 // de comandos, y (2) tengan valor por defecto.
87 //
88 for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op) {
89 #define OPTION_NAME(op) \
90     (op->short_name ? op->short_name : op->long_name)
91     if (op->flags & OPT_SEEN)
92         continue;
93     if (op->flags & OPT_MANDATORY) {
94         cerr << "Option "
95             << "-"
96             << OPTION_NAME(op)
97             << " is mandatory."
98             << "\n";
99         exit(1);
100     }
101     if (op->def_value == 0)
102         continue;
103     op->parse(string(op->def_value));
104 }
105
106 int
107 cmdline::do_long_opt(const char *opt, const char *arg)
108 {
109     // Recorremos la tabla de opciones, y buscamos la
110     // entrada larga que se corresponda con la opción de
111     // línea de comandos. De no encontrarse, indicamos el
112     // error.
113     //
114     for (option_t *op = option_table; op->long_name != 0; ++op) {
115         if (string(opt) == string(op->long_name)) {
116             // Marcamos esta opción como usada en
117             // forma explícita, para evitar tener
118             // que inicializarla con el valor por
119             // defecto.
120             //
121             op->flags |= OPT_SEEN;
122
123             if (op->has_arg) {
124                 // Como se trata de una opción
125                 // con argumento, verificamos que
126                 // el mismo haya sido provisto.
127                 //
128                 if (arg == 0) {
129                     cerr << "Option requires argument: "
130                         << "-"
131                         << opt
132                         << "\n";
133                     exit(1);
134                 }
135                 op->parse(string(arg));
136                 return 1;
137             } else {
138                 // Opción sin argumento.

```

```

140         //
141         op->parse(string(""));
142         return 0;
143     }
144 }
145
146 // Error: opción no reconocida. Imprimimos un mensaje
147 // de error, y finalizamos la ejecución del programa.
148 //
149 cerr << "Unknown option: "
150        << "--"
151        << opt
152        << "."
153        << endl;
154 exit(1);
155
156 // Algunos compiladores se quejan con funciones que
157 // lógicamente no pueden terminar, y que no devuelven
158 // un valor en esta última parte.
159 //
160 return -1;
161 }
162
163 int
164 cmdline::do_short_opt(const char *opt, const char *arg)
165 {
166     option_t *op;
167
168     // Recorremos la tabla de opciones, y buscamos la
169     // entrada corta que se corresponda con la opción de
170     // línea de comandos. De no encontrarse, indicamos el
171     // error.
172     //
173     for (op = option_table; op->short_name != 0; ++op) {
174         if (string(opt) == string(op->short_name)) {
175             // Marcamos esta opción como usada en
176             // forma explícita, para evitar tener
177             // que inicializarla con el valor por
178             // defecto.
179             //
180             op->flags |= OPT_SEEN;
181
182             if (op->has_arg) {
183                 // Como se trata de una opción
184                 // con argumento, verificamos que
185                 // el mismo haya sido provisto.
186                 //
187                 if (arg == 0) {
188                     cerr << "Option requires argument: "
189                            << "--"
190                            << opt
191                            << "\n";
192                     exit(1);
193                 }
194                 op->parse(string(arg));
195                 return 1;
196             } else {
197                 // Opción sin argumento.
198                 //
199                 op->parse(string(""));
200                 return 0;
201             }
202         }
203     }
204
205     // Error: opción no reconocida. Imprimimos un mensaje
206     // de error, y finalizamos la ejecución del programa.
207     //
208     cerr << "Unknown option: "

```

```
210         << "-"
211         << opt
212         << "."
213         << endl;
214     exit(1);
215
216     // Algunos compiladores se quejan con funciones que
217     // lógicamente no pueden terminar, y que no devuelven
218     // un valor en esta última parte.
219     //
220     return -1;
221 }
```

Listing 9: sha256.h

```

1  #ifndef SHA256_H
2  #define SHA256_H
3  #include <string>
4
5  class SHA256
6  {
7  protected:
8      typedef unsigned char uint8;
9      typedef unsigned int uint32;
10     typedef unsigned long long uint64;
11
12     const static uint32 sha256_k[];
13     static const unsigned int SHA224_256_BLOCK_SIZE = (512/8);
14 public:
15     void init();
16     void update(const unsigned char *message, unsigned int len);
17     void final(unsigned char *digest);
18     static const unsigned int DIGEST_SIZE = ( 256 / 8);
19
20 protected:
21     void transform(const unsigned char *message, unsigned int block_nb);
22     unsigned int m_tot_len;
23     unsigned int m_len;
24     unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
25     uint32 m_h[8];
26 };
27
28 std::string sha256(std::string input);
29
30 #define SHA2_SHFR(x, n) ((x >> n))
31 #define SHA2_ROTTR(x, n) ((x >> n) | (x << ((sizeof(x) << 3) - n)))
32 #define SHA2_ROTL(x, n) ((x << n) | (x >> ((sizeof(x) << 3) - n)))
33 #define SHA2_CH(x, y, z) ((x & y) ^ (~x & z))
34 #define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
35 #define SHA256_F1(x) (SHA2_ROTTR(x, 2) ^ SHA2_ROTTR(x, 13) ^ SHA2_ROTTR(x, 22))
36 #define SHA256_F2(x) (SHA2_ROTTR(x, 6) ^ SHA2_ROTTR(x, 11) ^ SHA2_ROTTR(x, 25))
37 #define SHA256_F3(x) (SHA2_ROTTR(x, 7) ^ SHA2_ROTTR(x, 18) ^ SHA2_SHFR(x, 3))
38 #define SHA256_F4(x) (SHA2_ROTTR(x, 17) ^ SHA2_ROTTR(x, 19) ^ SHA2_SHFR(x, 10))
39 #define SHA2_UNPACK32(x, str) \
40 { \
41     *((str) + 3) = (uint8) ((x) >> 24); \
42     *((str) + 2) = (uint8) ((x) >> 16); \
43     *((str) + 1) = (uint8) ((x) >> 8); \
44     *((str) + 0) = (uint8) ((x) >> 0); \
45 }
46 #define SHA2_PACK32(str, x) \
47 { \
48     *((x) + 3) = ((uint32) *((str) + 3) << 24) \
49     | ((uint32) *((str) + 2) << 16) \
50     | ((uint32) *((str) + 1) << 8) \
51     | ((uint32) *((str) + 0) << 0); \
52 }
53 #endif

```

Listing 10: sha256.cpp

```

1  #include <cstring>
2  #include <fstream>
3  #include "sha256.h"
4
5  const unsigned int SHA256::sha256_k[64] = //UL = uint32
6      {0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
7        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
8        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
9        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
10       0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
11       0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
12       0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
13       0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
14       0x27b70a85, 0x2e1b2138, 0x4d2c6dfe, 0x53380d13,
15       0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
16       0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
17       0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
18       0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
19       0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
20       0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
21       0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};
22
23 void SHA256::transform(const unsigned char *message, unsigned int block_nb)
24 {
25     uint32 w[64];
26     uint32 wv[8];
27     uint32 t1, t2;
28     const unsigned char *sub_block;
29     int i;
30     int j;
31     for (i = 0; i < (int) block_nb; i++) {
32         sub_block = message + (i << 6);
33         for (j = 0; j < 16; j++) {
34             SHA2_PACK32(&sub_block[j << 2], &w[j]);
35         }
36         for (j = 16; j < 64; j++) {
37             w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[
j - 16];
38         }
39         for (j = 0; j < 8; j++) {
40             wv[j] = m_h[j];
41         }
42         for (j = 0; j < 64; j++) {
43             t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
+ sha256_k[j] + w[j];
44             t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
45             wv[7] = wv[6];
46             wv[6] = wv[5];
47             wv[5] = wv[4];
48             wv[4] = wv[3] + t1;
49             wv[3] = wv[2];
50             wv[2] = wv[1];
51             wv[1] = wv[0];
52             wv[0] = t1 + t2;
53         }
54         for (j = 0; j < 8; j++) {
55             m_h[j] += wv[j];
56         }
57     }
58 }
59
60 void SHA256::init()
61 {
62     m_h[0] = 0x6a09e667;
63     m_h[1] = 0xbb67ae85;
64     m_h[2] = 0x3c6ef372;
65     m_h[3] = 0xa54ff53a;
66     m_h[4] = 0x510e527f;

```

```

68     m_h[5] = 0x9b05688c;
69     m_h[6] = 0x1f83d9ab;
70     m_h[7] = 0x5be0cd19;
71     m_len = 0;
72     m_tot_len = 0;
73 }
74
75 void SHA256::update(const unsigned char *message, unsigned int len)
76 {
77     unsigned int block_nb;
78     unsigned int new_len, rem_len, tmp_len;
79     const unsigned char *shifted_message;
80     tmp_len = SHA224_256_BLOCK_SIZE - m_len;
81     rem_len = len < tmp_len ? len : tmp_len;
82     memcpy(&m_block[m_len], message, rem_len);
83     if (m_len + len < SHA224_256_BLOCK_SIZE) {
84         m_len += len;
85         return;
86     }
87     new_len = len - rem_len;
88     block_nb = new_len / SHA224_256_BLOCK_SIZE;
89     shifted_message = message + rem_len;
90     transform(m_block, 1);
91     transform(shifted_message, block_nb);
92     rem_len = new_len % SHA224_256_BLOCK_SIZE;
93     memcpy(m_block, &shifted_message[block_nb << 6], rem_len);
94     m_len = rem_len;
95     m_tot_len += (block_nb + 1) << 6;
96 }
97
98 void SHA256::final(unsigned char *digest)
99 {
100     unsigned int block_nb;
101     unsigned int pm_len;
102     unsigned int len_b;
103     int i;
104     block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
105                     < (m_len % SHA224_256_BLOCK_SIZE)));
106     len_b = (m_tot_len + m_len) << 3;
107     pm_len = block_nb << 6;
108     memset(m_block + m_len, 0, pm_len - m_len);
109     m_block[m_len] = 0x80;
110     SHA2_UNPACK32(len_b, m_block + pm_len - 4);
111     transform(m_block, block_nb);
112     for (i = 0; i < 8; i++) {
113         SHA2_UNPACK32(m_h[i], &digest[i << 2]);
114     }
115 }
116
117 std::string sha256(std::string input)
118 {
119     unsigned char digest[SHA256::DIGEST_SIZE];
120     memset(digest, 0, SHA256::DIGEST_SIZE);
121
122     SHA256 ctx = SHA256();
123     ctx.init();
124     ctx.update((unsigned char*)input.c_str(), input.length());
125     ctx.final(digest);
126
127     char buf[2*SHA256::DIGEST_SIZE+1];
128     buf[2*SHA256::DIGEST_SIZE] = 0;
129     for (unsigned int i = 0; i < SHA256::DIGEST_SIZE; i++)
130         sprintf(buf+i*2, "%02x", digest[i]);
131     return std::string(buf);
132 }

```

6 Enunciado

75.04/95.12 Algoritmos y Programación II

Trabajo práctico 0: programación C++

Universidad de Buenos Aires - FIUBA
Segundo cuatrimestre de 2020

1. Objetivos

Ejercitar conceptos básicos de programación C++, implementando un programa y su correspondiente documentación que resuelva el problema descripto más abajo.

2. Alcance

Este Trabajo Práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado a través del campus virtual, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la Sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

4. Descripción

Bitcoin es, posiblemente, la criptomoneda más importante de la actualidad. Los trabajos prácticos de este cuatrimestre están destinados a comprender los detalles técnicos más relevantes detrás de Bitcoin –en particular, la tecnología de **blockchain**. Para ello, trabajaremos con **ALGOCHAIN**, una simplificación de la blockchain orientada a capturar los conceptos esenciales de la tecnología.

En este primer acercamiento al problema, nos abocaremos a leer y procesar **transacciones** y ensamblar un **bloque** a partir de estas. Estos conceptos serán debidamente introducidos en la Sección 4.1, donde daremos una breve introducción a Bitcoin y blockchain. Una vez hecho esto, presentaremos la **ALGOCHAIN** en la Sección 4.2, destacando al mismo tiempo las similitudes y diferencias más importantes con la blockchain propiamente dicha. Las tareas a realizar en el presente trabajo se detallan en la Sección 4.3.

4.1. Introducción a Bitcoin y blockchain

Una *criptomoneda* es un activo digital que actúa como medio de intercambio utilizando tecnología criptográfica para asegurar la autenticidad de las transacciones. Bitcoin es, tal vez, la criptomoneda más importante en la actualidad. Propuesta en 2009 por una persona (o grupo de personas) bajo el seudónimo *Satoshi Nakamoto* [2], se caracterizó por ser la primera criptomoneda descentralizada que propuso una solución al problema de *double-spending* sin involucrar una tercera parte de confianza¹. La idea esencial (y revolucionaria) que introdujo Bitcoin se basa en un registro descentralizado de todas las transacciones procesadas en el que cualquiera puede asentar operaciones. Este registro, replicado y distribuido en cada nodo de la red, se conoce como **blockchain**.

La blockchain no es otra cosa que una lista enlazada de *bloques*². Los bloques agrupan *transacciones* y son la unidad básica de información de la blockchain (i.e., son los nodos de la lista). Cuando un usuario introduce una nueva transacción t en la red, las propiedades de la blockchain garantizan una detección eficiente de cualquier otra transacción que extraiga los fondos de la misma operación referenciada por t . En caso de que esto sucediera, se considera que el usuario está intentando hacer *double-spending* y la transacción es consecuentemente invalidada por los nodos de la red.

En lo que sigue describiremos los conceptos más importantes detrás de la blockchain. La Figura 1 provee un resumen visual de todo estos conceptos en el marco de la ALGOCHAIN.

4.1.1. Funciones de hash criptográficas

Una *función de hash criptográfica* es un algoritmo matemático que toma una cantidad arbitraria de bytes y computa una tira de bytes de una longitud fija (en adelante, un *hash*). Es importante que dichas funciones sean *one-way*, en el sentido de que sea computacionalmente inviable hacer una “ingeniería reversa” sobre la salida para reconstruir una posible entrada. Otra propiedad que suelen tener dichas funciones es un *efecto avalancha* en el que cambios incluso en bits aislados de la entrada derivan en hashes significativamente diferentes.

Bitcoin emplea la función de hash SHA256, ampliamente utilizada en una gran variedad de protocolos de autenticación y encriptación [3]. Esta genera una salida de 32 bytes que usualmente se representa mediante 64 dígitos hexadecimales. A modo de ejemplo, el valor de $\text{SHA256}(\text{'Sarasa.'})$ es

9c231858fa5fef160c1e7ecfa333df51e72ec04e9c550a57c59f22fe8bb10df2

4.1.2. Direcciones y firmas digitales

Una *dirección* de Bitcoin es básicamente un hash de 160 bits de la clave pública de un usuario. Mediante algoritmos criptográficos asimétricos, los usuarios pueden generar pares de claves mutuamente asociadas (pública y privada). La *clave privada* se emplea para *firmar*

¹El doble gasto (o *double-spending*) ocurre cuando el emisor del dinero crea más de una transacción a partir de una misma operación previa. Naturalmente, sólo una de las nuevas transacciones debería ser válida puesto que, de lo contrario, el emisor estaría multiplicando dinero.

²Técnicamente, la blockchain es más bien un árbol de bloques, pero esto será abordado en el contexto del siguiente trabajo práctico.

los mensajes que desean transmitirse. Cualquier receptor puede luego verificar que la firma es válida utilizando la *clave pública* asociada. Esta clase de métodos criptográficos ofrecen garantías de que es computacionalmente difícil reconstruir la clave privada a partir de la información públicamente disponible.

4.1.3. Transacciones

Una *transacción* en Bitcoin está definida por una lista de entradas (*inputs*) y otra de salidas (*outputs*). Un *output* se representa a través de un par (*value*, *addr*), donde *value* indica la cantidad de bitcoins que recibirá el destinatario y *addr* es el hash criptográfico de la clave pública del destinatario. Por otro lado, un *input* puede entenderse como una tupla (*tx_id*, *idx*, *key*, *sig*) tal que:

- *tx_id* indica el hash de una transacción previa de la que esta nueva transacción toma fondos,
- *idx* es un índice dentro de la secuencia de *outputs* de dicha transacción (los fondos de este *input*, luego, provienen de dicho *output*),
- *key* es la clave pública asociada a tal *output*, y
- *sig* es la firma digital del hash de la transacción usando la clave privada asociada a la clave pública del *output*.

En consecuencia, cada *input* hace referencia a un *output* anterior en la blockchain (los campos *tx_id* y *idx* suelen agruparse en una estructura común bajo el nombre de *outpoint*). Para verificar que el uso de dicho *output* es legítimo, se calcula el hash de la clave pública y se verifica que sea igual a la que figura en el *output* utilizado. Luego, basta con verificar la firma digital con esa clave pública para asegurar la autenticidad de la operación.

Para garantizar la validez de una transacción, es importante verificar no sólo que cada *input* es válido sino también que la suma de los *outputs* referenciados sea mayor o igual que la suma de los *outputs* de la transacción. La diferencia entre ambas sumas, en caso de existir, es lo que se conoce como *transaction fee*. Este valor puede ser reclamado por quien agrega la transacción a la blockchain (como retribución por suministrar poder de cómputo para realizar el minado de un nuevo bloque).

Naturalmente, una vez que un *output* de una transacción haya sido utilizado, este no podrá volver a utilizarse en el futuro. En otras palabras, cada nueva transacción sólo puede referenciar *outputs* que no fueron utilizados previamente. Estos últimos se conocen como *unspent transaction outputs* (UTXOs).

4.1.4. Bloques

Toda transacción de Bitcoin pertenece necesariamente a un *bloque*. Cada bloque está integrado por un encabezado (*header*) y un cuerpo (*body*). En el header se destaca la siguiente información:

- El hash del bloque antecesor en la blockchain (*prev_block*),

- El hash de todas las transacciones incluidas en el bloque (`txns_hash`),
- La *dificultad* con la cual este bloque fue ensamblado (`bits`), y
- Un campo en el que se puede poner datos arbitrarios, permitiendo así alterar el hash resultante (`nonce`).

El cuerpo de un bloque, por otro lado, incluye la cantidad total de transacciones (`txn_count`) seguido de la secuencia conformada por dichas transacciones (`txns`).

4.1.5. Minado de bloques

Para que un bloque sea válido y pueda en consecuencia ser aceptado por la red de Bitcoin, debe contar con una prueba de trabajo (*proof-of-work*) que debe ser difícil de calcular y, en simultáneo, fácil de verificar. El mecanismo detrás de este proceso se conoce como *minado*. Las entidades encargadas de agrupar transacciones y ensamblar bloques válidos son los *mineros*.

Los mineros obtienen una recompensa en bitcoins cuando agregan un bloque a la blockchain. Esto último se logra calculando la *proof-of-work* del nuevo bloque, lo cual a su vez se realiza con poder de cómputo. La *proof-of-work* de un bloque consiste en un hash $h = \text{SHA256}(\text{SHA256}(\text{header}))$ tal que su cantidad de ceros en los bits más significativos es mayor o igual que un valor derivado del campo `bits` del header del bloque. A los efectos prácticos, consideraremos que, si el campo `bits` indica un valor d , la cantidad de ceros en los bits más significativos de h debe ser $\geq d$.

El esfuerzo necesario para ensamblar un bloque que cumpla con la dificultad de la red crece exponencialmente con la cantidad de ceros requerida. Esto se debe a que agregar un cero extra a la dificultad disminuye a la mitad la cantidad de hashes que cumplan con dicha restricción. No obstante esto, para verificar que un bloque cumple con esta propiedad, basta con computar dos veces la función de hash SHA256. De esta forma, se puede comprobar fácilmente que un minero realizó una cierta cantidad de trabajo para hallar un bloque válido.

4.2. Algochain: la blockchain de Algoritmos II

La blockchain simplificada con la que estaremos trabajando a lo largo del cuatrimestre es la ALGOCHAIN. Al igual que la blockchain, la ALGOCHAIN se compone de bloques que agrupan transacciones. A su vez, las transacciones constan de una secuencia de *inputs* y otra de *outputs* que siguen los mismos lineamientos esbozados más arriba. La Figura 1 muestra un esquema de alto nivel de la ALGOCHAIN.

4.2.1. Direcciones

Una de las diferencias más importantes con la blockchain radica en una simplificación intencional del proceso de verificación y validación de direcciones al momento de procesar las transacciones: la ALGOCHAIN no utiliza firmas digitales ni claves públicas. En su lugar, tanto los *inputs* como los *outputs* de las transacciones referencian directamente direcciones de origen y destino de los fondos, respectivamente. Esta *dirección* la interpretaremos como un hash SHA256 de una cadena de caracteres arbitraria que simbolice la dirección propiamente dicha

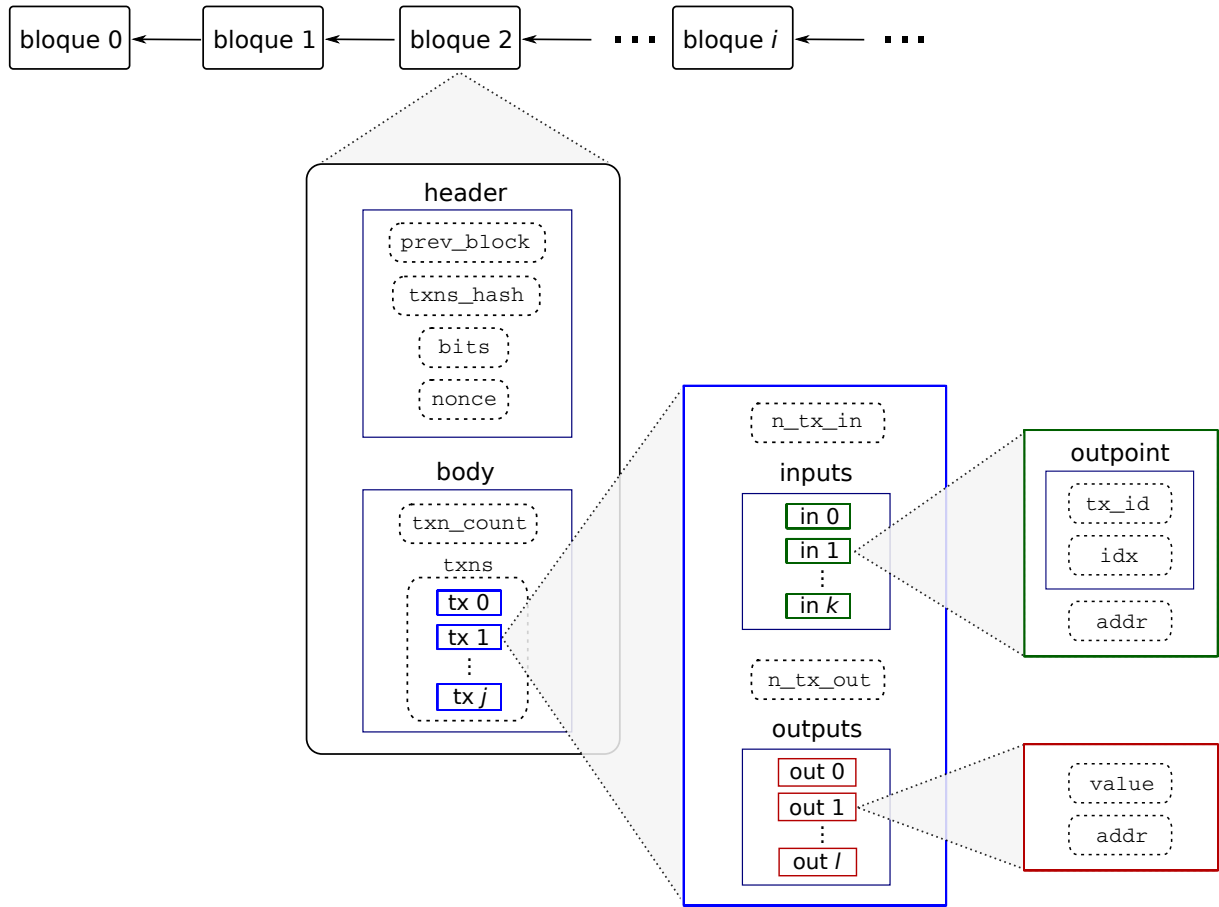


Figura 1: Esquema de alto nivel de la ALGOCHAIN

del usuario. A la hora de procesar una nueva transacción t , simplemente deberá validarse que la dirección `addr` de cada *input* de t coincida exactamente con el valor del `addr` especificado en el *output* referenciado en dicho *input*.

A modo de ejemplo, si la dirección real de un usuario de nuestra ALGOCHAIN fuese Segurola y Habana, los campos `addr` de las transacciones que involucren a dicho usuario deberían contener el valor `addr = SHA256('Segurola y Habana')`, que equivale a

485c8c85be20ebb6a9f6dd586b0f9eb6163aa0db1c6e29185b3c6cd1f7b15e9e

4.2.2. Hashes de bloques y transacciones

Tal como ocurre en blockchain, y como explicamos en la Sección 4.1, en ALGOCHAIN identificamos unívocamente bloques y transacciones mediante hashes SHA256 dobles.

El campo `prev_block` del header de un bloque b indica el hash del bloque antecesor b' en la ALGOCHAIN. De este modo, `prev_block = SHA256(SHA256(b'))`. Dicho hash lo calcularemos sobre una concatenación secuencial de todos los campos de b' respetando exactamente el formato de bloque que describiremos en la Sección 4.4.

De forma análoga, el campo `tx_id` de los *inputs* de las transacciones lo calcularemos con un doble hash SHA256 sobre una concatenación de todos los campos de la transacción correspondiente.

Finalmente, el campo `txns_hash` del header de un bloque b contendrá también un doble hash SHA256 de todas las transacciones incluidas en b . En el contexto de este trabajo práctico, dicho hash lo calcularemos sobre una concatenación de todas las transacciones respetando exactamente el formato que describiremos en la Sección 4.4. En otras palabras, dadas las transacciones $t_0, t_1 \dots, t_j$ del bloque b ,

$$\text{txns_hash} = \text{SHA256}(\text{SHA256}(t_0 t_1 \dots t_j))$$

4.3. Tareas a realizar

Para apuntalar los objetivos esenciales de este trabajo práctico, esbozados en la Sección 1, deberemos escribir un programa que reciba transacciones por un *stream* de entrada y ensamble un bloque con todas ellas una vez finalizada la lectura. Dicho bloque deberá escribirse en un *stream* de salida. De este modo, al estar trabajando con único bloque, por convención dejaremos fijo el valor del campo `prev_block` en su header. Dicho campo debe instanciarse en

```
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

Naturalmente, nuestros bloques deben satisfacer los requisitos de validez delineados en la Sección 4.1.5. En particular, nos interesa exhibir la correspondiente *proof-of-work* para poder reclamar las eventuales recompensas derivadas del minado. Para ello, nuestros programas recibirán como parámetro el valor de la *dificultad* d esperada. En otras palabras, debemos garantizar que la cantidad de ceros en los bits más significativos de nuestro hash h es $\geq d$, siendo $h = \text{SHA256}(\text{SHA256}(\text{header}))$. Recordar que, en caso de no encontrar un hash h válido, es posible intentar sucesivas veces modificando el campo `nonce` del header del bloque (la Sección 4.4 describe en detalle el formato de dicho header). Este campo puede instanciarse con valores numéricos arbitrarios tantas veces como sea necesario hasta dar con un hash válido.

Para simplificar el proceso de desarrollo, la cátedra suministrará código C++ para calcular hashes SHA256.

4.4. Formatos de la Algochain

En esta Sección detallaremos el formato de las transacciones y bloques de la ALGOCHAIN. Tener en cuenta que es sumamente importante **respetar de manera estricta** este formato. Mostraremos algunos ejemplos concretos en la Sección 4.6.

4.4.1. Transacciones

Toda transacción de la ALGOCHAIN debe satisfacer el siguiente formato:

- Empieza con una línea que contiene el campo entero `n_tx_in`, que indica la cantidad total de *inputs*.

- Luego siguen los *inputs*, uno por línea. Cada *input* consta de tres campos separados entre sí por un único espacio:
 - *tx_id*, el hash de la transacción de donde este *input* toma fondos,
 - *idx*, un valor entero no negativo que sirve de índice sobre la secuencia de *outputs* de la transacción con hash *tx_id*, y
 - *addr*, la dirección de origen de los fondos (que debe coincidir con la dirección del *output* referenciado).
- Luego de la secuencia de *inputs*, sigue una línea con el campo entero *n_tx_out*, que indica la cantidad total de *outputs* en la transacción.
- Las *n_tx_out* líneas siguientes contienen la secuencia de *outputs*, uno por línea. Cada *output* consta de los siguientes campos, separados por un único espacio:
 - *value*, un número de punto flotante que representa la cantidad de Algos a transferir en este *output*, y
 - *addr*, la dirección de destino de tales fondos.

4.4.2. Bloques

Como indicamos en la Sección 4.1.4, todo bloque arranca con un header. El formato de nuestros headers es el siguiente:

- El primer campo es *prev_block*, que contiene el hash del bloque completo que antecede al bloque actual en la ALGOCHAIN.
- Luego sigue el campo *txns_hash*, que contiene el hash de todas las transacciones incluidas en el bloque. El cálculo de este hash debe realizarse de acuerdo a las instrucciones de la Sección 4.2.2.
- A continuación sigue el campo *bits*, un valor entero positivo que indica la dificultad con la que fue minada este bloque.
- El último campo del header es el *nonce*, un valor entero no negativo que puede contener valores arbitrarios. El objetivo de este campo es tener un espacio de prueba modificable para poder generar hashes sucesivos hasta satisfacer la dificultad del minado.

Todos estos campos deben aparecer en líneas independientes. En la línea inmediatamente posterior al *nonce* comienza la información del body del bloque:

- La primera línea contiene el campo *txn_count*, un valor entero positivo que indica la cantidad total de transacciones incluidas en el bloque.
- A continuación siguen una por una las *txn_count* transacciones. Todas ellas deben respetar el formato de transacción de la Sección anterior.

4.5. Interfaz

La interacción con nuestros programas se dará a través de la línea de comando. Las opciones a implementar en este trabajo práctico son las siguientes:

- `-d`, o `--difficulty`, que indica la dificultad esperada d del minado del bloque. En otras palabras, el hash $h = \text{SHA256}(\text{SHA256}(\text{header}))$ debe ser tal que la cantidad de ceros en sus bits más significativos sea $\geq d$. Esta opción es de carácter obligatorio (i.e., el programa no puede continuar en su ausencia).
- `-i`, o `--input`, que permite controlar el stream de entrada de las transacciones. El programa deberá recibir las transacciones a partir del archivo con el nombre pasado como argumento. Si dicho argumento es `"-"`, el programa las leerá de la entrada standard, `std::cin`.
- `-o`, o `--output`, que permite direccionar la salida al archivo pasado como argumento o, de manera similar a la anterior, a la salida standard `-std::cout` si el argumento es `"-"`.

4.6. Ejemplos

Consideremos la siguiente transacción t :

```
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

Por una cuestión de espacio, los hashes involucrados en estos ejemplos aparecen representados por sus últimos 8 bytes. De este modo, el hash `tx_id` del *input* de t y las direcciones *addr* referenciadas en el *input* y en el *output* son, respectivamente,

```
26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779
f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
```

Esta transacción consta de un único *input* y un único *output*. El *input* toma fondos de alguna supuesta transacción t' cuyo hash es `48df0779`. En particular, los fondos provienen del tercer *output* de t' (observar que `idx` es 2). La dirección de origen de los fondos es `d4cc51bb`. Por otra parte, el *output* de t deposita 250,5 Algos en la dirección `842f33e9`.

Supongamos ahora que contamos con un archivo de transacciones que contiene la información de t :

```
$ cat txns.txt
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

La siguiente invocación solicita ensamblar un nuevo bloque con esta transacción:

```
$ ./tp0 -i txns.txt -o block.txt -d 3
```

Notar que el bloque debe escribirse al archivo `block.txt`. Además, la dificultad de minado sugerida es 3: esto nos dice que el hash del header de nuestro bloque debe comenzar con 3 o más bits nulos. Una posible salida podría ser la siguiente:

```
$ cat block.txt
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
155dc94b29dce95bb2f940cdd2d7e0bce66dca9370c3ed96d50a30b3d84f8c4c
3
12232
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

Observar que el valor del nonce es 12232. Si bien dicho nonce permite encontrar un hash del header satisfactorio, esta elección por supuesto no es única. El hash del header, bajo esta elección, resulta

```
045b22553f86219b1ecb68bc34a623ecff7fe1807be806a3ccfa9f1b3df5cfc0
```

Como puede verse, el hash comienza con cuatro bits nulos.

Un ejemplo aún más simple (y curioso) consiste en invocar nuestros programas con una entrada vacía:

```
$ touch empty.txt
$ ./tp0 -i empty.txt -d 3
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f5a6adc36f8e6
3
59329
0
```

En primer lugar, notemos que, al no especificar un *stream* de salida, el programa dirige la escritura del bloque a la salida estándar. El bloque construido, si bien no incluye ninguna transacción, contiene información válida en su header. Notar que el campo `txns_hash` se calcula en este caso a partir del doble hash SHA256 de una cadena de caracteres vacía.

4.7. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

5. Informe

El informe deberá incluir, como mínimo:

- Una carátula que incluya los nombres de los integrantes y el listado de todas las entregas realizadas hasta ese momento, con sus respectivas fechas.
- Documentación relevante al diseño e implementación del programa.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C++.
- Este enunciado.

6. Fechas

La última fecha de entrega es el **jueves 12 de noviembre de 2020**.

Referencias

- [1] Wikipedia, "Bitcoin Wiki." https://en.bitcoin.it/wiki/Main_Page.
- [2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.
- [3] Wikipedia, "SHA-2." <https://en.wikipedia.org/wiki/SHA-2>.