Fix Branch

CMPUT 229

University of Alberta

Consider the **original** program below

RISC-V Assembly

```
1  foo:
2      beq  t0 t1 target
3      addi t0 t0 1
4  target:
5      addi t1 t1 1
```

RARS

Address	Code	Basic
0x00400024	0x00628463	beq x5, x6, 0x00000008
0x00400028	0x00128293	addi x5, x5, 1
0x0040002c	**0x00130313**	**addi x6, x6, 1**

Address of branch target is `0x0040002c`

After inserting three instructions, we obtain the **modified** program.

RARS

Address	Code	Basic
0x00400024	0x00628463	beq x5, x6, 0x00000008
0x00400028	0x00128293	addi x5, x5, 1
0x0040002c	**0x00130313**	**addi x6, x6, 1**
0x00400030	0x00130393	addi x7, x6, 1
0x00400034	0x00138e13	addi x28, x7, 1
0x00400038	**0x00130313**	**addi x6, x6, 1**

Branch offset did not change

Address of branch target is changed to 0x00400038

Task

Branch offset did not change

RARS

Address	Code	Basic
0x00400024	0x00628463	beq x5, x6, 0x00000008
0x00400028	0x00128293	addi x6, x6, 1
0x0040002c	0x00130393	addi x6, x6, 1
0x00400030	0x00130393	addi x7, x6, 1
0x00400034	0x00138e13	addi x28, x7, 1
0x00400038	**0x00130313**	**addi x6, x6, 1**

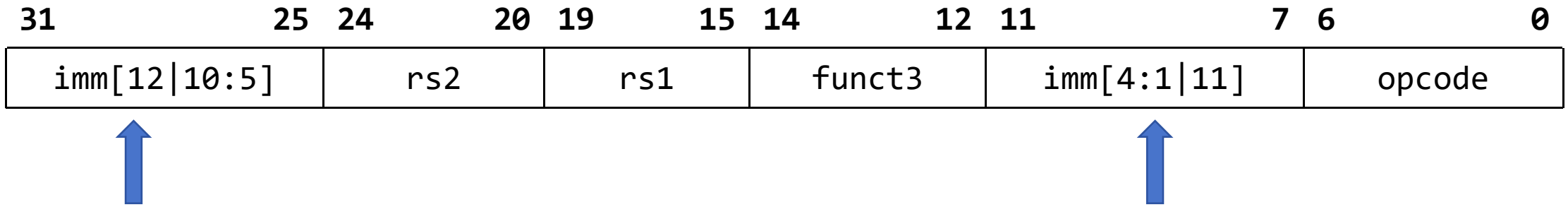
Adjust the offset such that the branch instruction branches to the same target label as it did in the original program

RISC-V Branch Instructions

```
beq         Branch EQual
bge         Branch Greater than or Equal
bgeu        Branch Greater than or Equal Unsigned
blt         Branch Less Than
bltu        Branch Less Than Unsigned
bne         Branch Not Equal
```

SB Type Format

All branch instructions are encoded in the SB Instruction Type Format



The relative offset is encoded in bits 7-11 & 25-31 of the instruction

SB Type Format - Limitations

- Only 13 bits to to specify the offset (the lowest bit of the immediate is always zero)
  - The distance between a branch instruction and its target is limited
- Inserting instructions may push the target beyond the reach of the branch instruction
  - Assume this will never the case in this lab

Jump Instructions

- Inserting instruction can also affect jump instructions
- You **DO NOT** need to adjust the immediate value of jump instructions in this lab

Fixing Branch Instructions

- A branch instruction only needs to be fixed if instructions are inserted between itself and the branch target
- To calculate the address of the branch target, T, follow the algorithm below:
  1. L ← immediate of the branch instruction
  2. A ← L << 1
  3. S ← sign extend A to 32 bits
  4. T ← S + PC
- Here, PC refers to the address that was used to load the instruction from memory

Insertion Points

- Address of the instruction (in the original program) before a block of added instructions

Insertion Points - Demonstration

Original Program

```
0x10010000    foo:    addi t0 zero 1
0x10010004            addi t1 zero 2
0x10010008            bne  t0 zero T1
0x1001000c            add  a7 t0 t1
0x10010010    T1:     jr   ra
```

Modified Program

```
foo:    addi t0 zero 1
        addi t1 zero 2
        bne  t0 zero T1
        add  a7 t0 t1
        add  t6 t3 t1
        and  t0 t3 t1
        or   t6 t3 t1
T1:     jr   ra
```

Insertion point: [0x1001000c]

Number of instructions inserted:

[3]

Terminologies

Different Types of Programs

- The **modified program** is obtained by inserting instructions into the **original program**.
- Branch instructions in the modified program must be fixed so that they jump to the same label as they did in the original program.
- The **fixed program** refers to the modified program with fixed branch instructions.

Instructions Array

- The **instructions array** of a program is an array of RISC-V instructions in binary representation
- The instructions in the array are the instructions in the program.

Insertion Points Array

- The **insertion points array** is an array of all the insertion points.
- The array must be sorted in ascending order by the value of the insertion points.

Insertions Array

- The **insertions array** is an array of natural numbers
- The `i`'th element in the array is the number of instructions inserted at the `i`'th insertion point in the insertion points array.

Branches Array

- The **branches array** contains the addresses of all the branch instructions in the original program.
- Elements in the array must be sorted in ascending order by the addresses of the branch instructions.

Targets Array

- The **targets array** contains the addresses of the target instructions for all the branch instructions in the branches array.
- The `i`'th element in the array is the address of the branch target for the `i`'th branch instruction in the branches array.

An Important Note

- All arrays are terminated by the sentinel value `0xFFFFFFFF`

Assignment

Students must implement all of the following functions

fixBranch

This function is the entry point of student's solution to this lab. It fixes branch instructions in the modified program such that they branch to the same target as they did in the original program. Writes the instructions of the fixed program program to a separate array.

Arguments:

a0: Pointer to the instructions array of the original program.

a1: Pointer to the instructions array of the modified program.

a2: Pointer to an empty array that is the same size as the instructions array of the modified program. This function must write the instructions of the fixed program to this array and terminate it with the sentinel value 0xFFFFFFFF.

Returns:

None

findInsertions

Finds all the insertion points and how many instructions were inserted after each insertion point

Arguments:

a0: Pointer to the instructions array of the original program.

a1: Pointer to the instructions array of the modified program.

Returns:

a0: Pointer to the insertion points array.

a1: Pointer to the insertions array.

findBranches

Finds all the branches and their respective targets in the original program

Arguments:

a0: Pointer to the instructions array of the original program.

Returns:

a0: Pointer to the branches array

a1: Pointer to the targets array

Resources

We provide the two Python progams:

- `binDecompiler.py`: a decompiler for RISC-V assembly.
- `hexToBin.py`: converts hexadecimal numbers in textual format to binary format and writes the results to a file. Expects one hexadecimal number per line.

Important Information

- The webpage and README contains the specifications and example usages of the two Python programs in the previous slide.
- The webpage also contains step-by-step instructions on how to create your own test cases and how to test you solution to the lab.

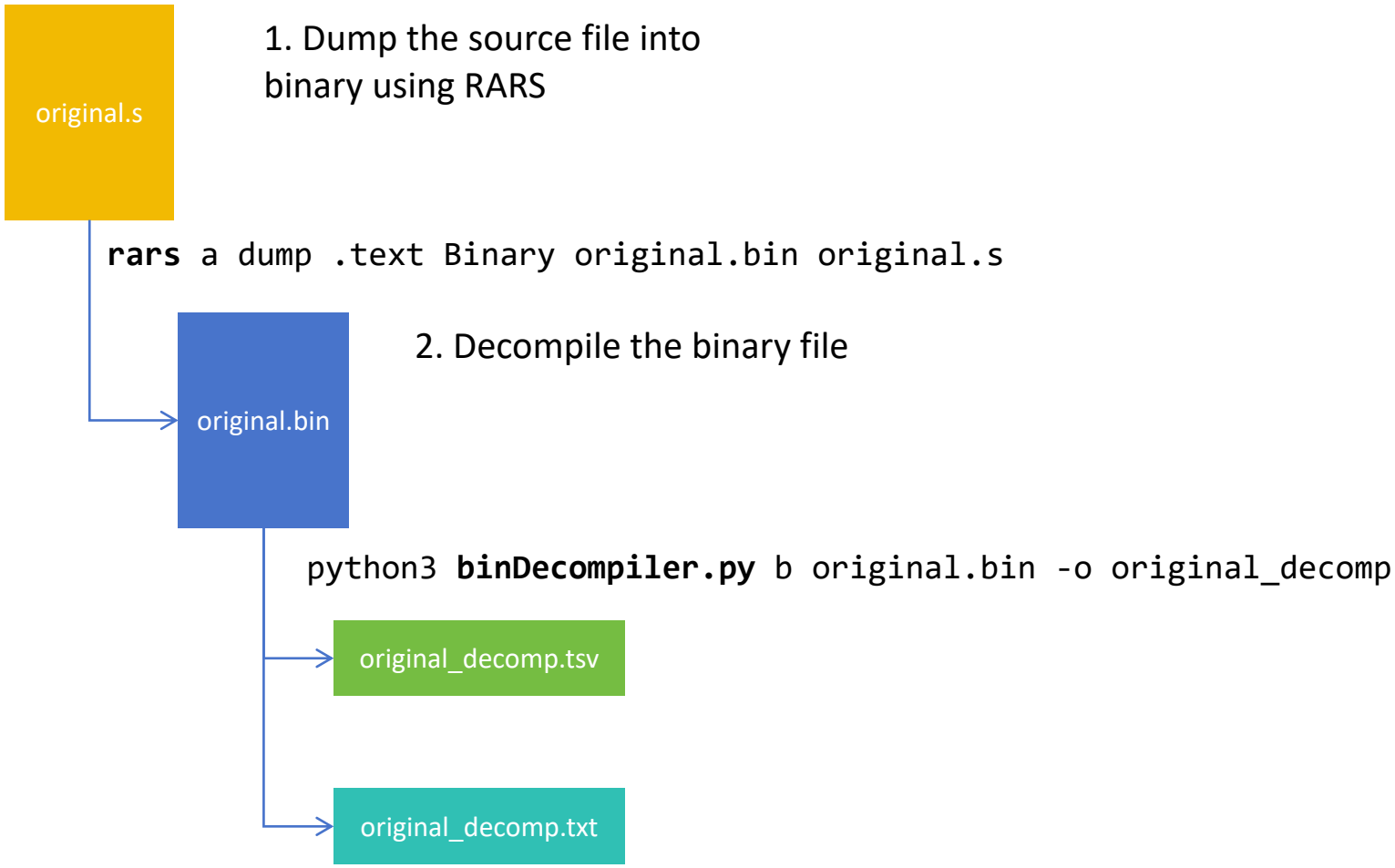
Testing your Lab

Input Guarantees

- The binary representation of the RISC-V programs are terminated by the sentinel value `0xFFFFFFFF`.
- There will be a maximum of 50 instructions inserted to the modified program
- There will be a maximum of 25 branches to fix.
- All instructions are valid RISC-V instructions.
- Neither branches nor jumps will ever be inserted to the original program to create the modified program.

original.s

Start with a RISC-V .s file that satisfies the constraints outlined in the Input Guarantees slide


original.s
1. Dump the source file into binary using RARS
rars a dump .text Binary original.bin original.s
original.bin
2. Decompile the binary file
python3 binDecompiler.py b original.bin -o original_decomp
original_decomp.tsv
original_decomp.txt
3. Create the modified program

Creating the Modified Program

- Option 1: Use a spreadsheet editor
  - Google Spreadsheets
  - Microsoft Excel
  - Some IDE’s
- Option 2: Use a plain text editor

Option 1

Open `original_decomp.tsv` with a spreadsheet editor

Address	Code	Source
0x00400024	0x00628463	beq x5, x6, 0x00000008
0x00400028	0x00128293	addi x5, x5, 1
0x0040002c	**0x00130313**	**addi x6, x6, 1**

Insert hexadecimal representation of valid RISC-V instructions in the ‘Code’ column

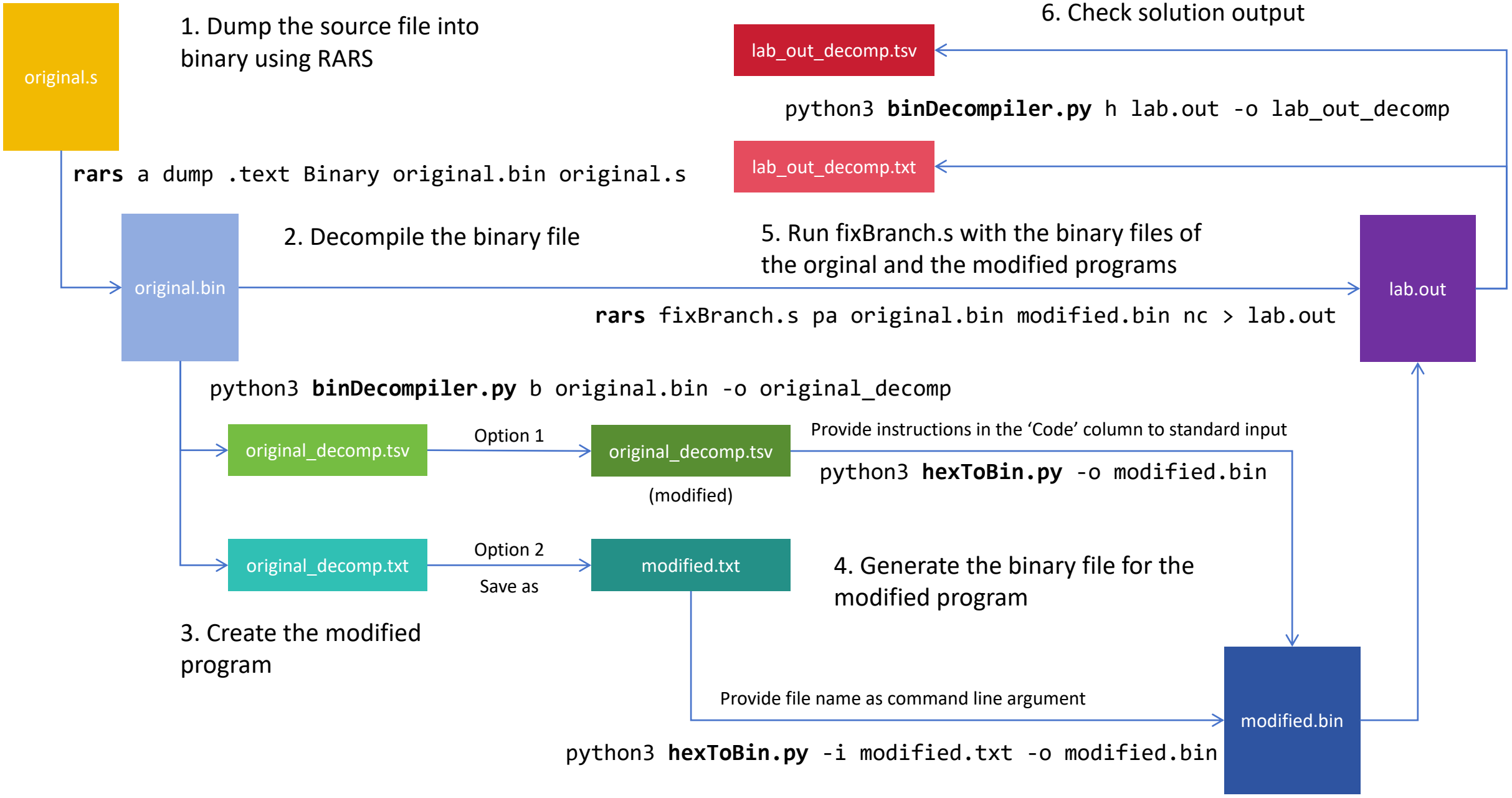
Address	Code	Source
0x00400024	0x00628463	beq x5, x6, 0x00000008
0x00400028	0x00128293	addi x5, x5, 1
	0x00128313	
	0x00130393	
	0x00138e13	
0x0040002c	**0x00130313**	**addi x6, x6, 1**

Option 2

Open `original_decomp.txt` with a plain text editor

```
0x00628463
0x00128293
0x00130313
```


```
0x00628463
0x00128293
0x00128313
0x00130393
0x00138e13
0x00130313
```


original.s
1. Dump the source file into binary using RARS
rars a dump .text Binary original.bin original.s
original.bin
2. Decompile the binary file
python3 binDecompiler.py b original.bin -o original_decomp
original_decomp.tsv
Option 1
original_decomp.tsv
(modified)
Provide instructions in the 'Code' column to standard input
python3 hexToBin.py -o modified.bin
original_decomp.txt
Option 2
Save as
modified.txt
3. Create the modified program
4. Generate the binary file for the modified program
Provide file name as command line argument
python3 hexToBin.py -i modified.txt -o modified.bin
modified.bin
5. Run fixBranch.s with the binary files of the orginal and the modified programs
rars fixBranch.s pa original.bin modified.bin nc > lab.out
lab.out
python3 binDecompiler.py h lab.out -o lab_out_decomp
lab_out_decomp.tsv
lab_out_decomp.txt
6. Check solution output