ECE26400 Programming Exercise #9

This exercise builds on PE08. It aims to familiarize you with linked lists and trees, which would be useful for PA05.

The main learning goals are:

1. How to create and manipulate linked lists and trees.
2. How to free the memory associated with a linked list and a tree.

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pe09_files.zip` using the following command:

```
unzip pe09_files.zip
```

The zip file `pe09_files.zip` contains a folder, named `PE09`, and a subfolder `examples` containing a few examples:

1. Please see Section 3 about the input files (`gophers`, `stone`, `empty`, `binary1`) and the corresponding output files.

In this exercise, you have to decide what `.h` file(s) to create and what `.c` file(s) to create. You must also create a `Makefile` that we will use to compile your source code into an executable called `pe09` using the command "`make pe09`".

2 Huffman coding

2.1 ASCII coding (and extended ASCII coding)

In ASCII coding, every character is encoded (represented) with the same number of bits (8-bits) per character, with numeric values ranging between 0 and 255 (inclusive). Since there are 256 different values that can be represented with 8-bits, there are potentially 256 different characters in the ASCII character set, as shown in the ASCII character table (and extended ASCII character table) available at http://www.asciitable.com/.

Character	ASCII value	8-bit binary value (Left is MSB)
`Space ' '`	32	00100000
`'e'`	101	01100101
`'g'`	103	01100111
`'h'`	104	01101000
`'o'`	111	01101111
`'p'`	112	01110000
`'r'`	114	01110010
`'s'`	115	01110011

Using ASCII encoding (8 bits per character) the 13-character stream `"go go gophers"` requires $13 \times 8 = 104$ bits.[1] The table to the right shows how the coding works. **From left to right, the binary bits for each character are ordered from the most significant bit (MSB) to the least significant bit (LSB).**

The given character stream would be written in a file as 13 bytes, represented by the following stream of bits:

```
    01100111 01101111 00100000 01100111 01101111 00100000 01100111 01101111 01110000
01101000 01100101 01110010 01110011
```

[1]We do not consider `NULL` character `'\0'` to be a part of this stream in this example.

2.2 A more efficient coding

There is a more efficient coding scheme that uses fewer bits. As there are only 8 different characters in `"go go gophers"`, we can use 3 bits to encode each of the 8 different characters. We may, for example, use the coding shown in the second table in the previous page (other 3-bit encodings are also possible). Again, we assume that **from left to right, the 3 bits for each character is ordered from the MSB to the LSB.**

Character	Code value	3-bit binary value (Left is MSB)
`'g'`	0	000
`'o'`	1	001
`'p'`	2	010
`'h'`	3	011
`'e'`	4	100
`'r'`	5	101
`'s'`	6	110
`Space ' '`	7	111

Now the character stream `"go go gophers"` would be encoded using a total of 39 bits instead of 104 bits. We can store that as five 8-bit bytes in a file as follows (**in each byte, left to right is MSB to LSB**):

`11001000 10010001 00100011 00011010 01101011.`

The first byte contains the 3 bits of `'g'` at the least significant positions, the 3 bits of `'o'` in the middle, and the less significant 2 bits of `Space`. The least significant bit of the second byte contains the most significant bit of `Space`. In positions of increasing significance, the second byte also contains `'g'`, `'o'`, and the least significant bit of `Space`. As five bytes contains 40 bits altogether, the most significant position of the last byte in the file is not used. **For the purpose of this course, all unused bits should be assigned the value of 0.**

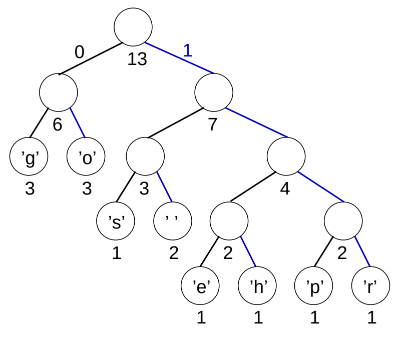
However, even in this improved coding scheme, we used the same number of bits to represent each character, regardless of how often the character appears in our character stream. Even more bits can be saved if we use fewer than three bits to encode characters like `'g'`, `'o'`, and `Space` that occur frequently and more than three bits to encode characters like `'e'`, `'h'`, `'p'`, `'r'`, and `'s'` that occur less frequently in `"go go gophers"`. This is the basic idea behind Huffman coding: use fewer bits for characters that occur more frequently. We'll see how this is done using a strictly binary tree that stores the characters as its leaf nodes, and whose root-to-leaf paths provide the bit sequences used to encode the characters.

A strictly binary tree is a binary tree where a node has either 0 or 2 child nodes. A node with 0 child nodes is a leaf node and a node with 2 child nodes is an internal (non-leaf) node.

2.3 More efficient coding using Huffman coding

Using a strictly binary tree for coding, all characters are stored at the leaves of a tree. A left-edge is numbered 0 and a right-edge is numbered 1. The code for any character/leaf node is obtained by following the root-to-leaf path and concatenating the 0's and 1's. The specific structure of the tree determines the coding of any leaf node using the 0/1 edge convention described. As an example, the tree to the right yields the coding shown in the table. **Unlike before, from left to right, the binary bits for each character in the table are ordered from LSB to MSB.**

Character	Huffman code (Left is LSB)
`'g'`	00
`'o'`	01
`'s'`	100
`Space ' '`	101
`'e'`	1100
`'h'`	1101
`'p'`	1110
`'r'`	1111



Using this coding, `"go go gophers"` can be encoded with 37 bits, two bits fewer than the 3-bit coding scheme. **From the LSB to the MSB, the bit stream from left to right** is as follows:

`00 01 101 00 01 101 00 01 1110 1101 1100 1111 100.`

Of course, we still have to use 5 bytes to store the 37 bits in a file. To show the bytes stored in the file, we insert | to delimit every 8 bits as follows:

`00 01 101 0|0 01 101 00| 01 1110 11|01 1100 11|11 100.`

Then, we write in the **convention of the LSB appearing on the right for each byte** as follows:

`01011000 00101100 11011110 11001110 00000111.`

In this case, **the three most significant bits of the last byte in the file are unused and they are assigned the value of 0.**

To decode a non-empty stream (from the least significant position to the most) that has been coded by the given tree, start at the root of the tree, and follow a left-branch if the next bit in the stream is a 0, and a right branch if the next bit in the stream is a 1. When you reach a leaf, write the character stored at the leaf, and start again at the top of the tree.

2.4 Building a Huffman coding tree

The coding tree in this example was constructed using an algorithm invented by David A. Huffman in 1952 when he was a Ph.D. student at MIT. We shall discuss how to construct the coding tree using Huffman's algorithm. We assume that associated with each character is a weight that is equal to the number of times the character occurs in a file. For example, in `"go go gophers"`, the characters `'g'` and `'o'` have weight 3, the `Space` has weight 2, and the other characters have weight 1.

When compressing a file, we first read the file and calculate these weights. Assume that all the character weights have been calculated. Huffman's algorithm assumes that we are building a single tree from a group (or forest) of trees. Initially, all the trees have a single node containing a character and the character's weight. Iteratively, a new tree is formed by picking two trees and making a new tree whose child nodes are the roots of the two trees. The weight of the new tree is the sum of the weights of the two sub-trees. This decreases the number of trees by one in each iteration. The process iterates until there is only one tree left. The algorithm is as follows:

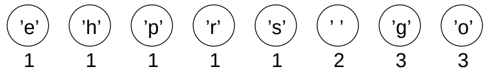
1. Begin with a forest of trees. All trees have just one node, with the weight of the tree equal to the weight of the character in the node. Characters that occur most frequently have the highest weights. Characters that occur least frequently have the smallest weights.

2. Repeat this step until there is only one tree:

   - Choose two trees with the smallest weights; call these trees T_1 and T_2.
   - Create a new tree whose root has a weight equal to the sum of the weights of T_1 and T_2, and whose left sub-tree is T_1 and whose right sub-tree is T_2.

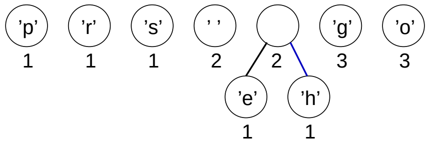
3. The single tree left after the previous step is Huffman's coding tree.

For the character stream `"go go gophers"`, we initially have the forest shown to the right. The nodes are shown with a weight that represents the number of times the node's character occurs in the given input character stream or input file.

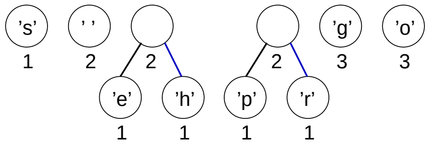


We remove two minimal nodes. There are five (root) nodes with the minimal weight of 1. In this implementation, we maintain a priority queue with items arranged according to their weights, i.e., the items are sorted. When two items have the same weight, a leaf node (i.e., a node associated with an ASCII character) is always ordered first. If both nodes are leaf nodes, they are ordered according to their ASCII coding (or numeric value); a node with a lower numeric value is ordered first in the priority queue. If both nodes are non-leaf nodes, they are ordered according to the creation times of the nodes; the node created earlier is ordered first in the priority queue.

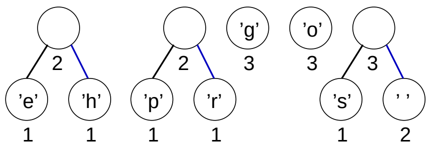
We always remove the first two items in the priority queue. Therefore, nodes for characters `'e'` and `'h'` are removed. We create a new tree whose root is weighted by the sum of the weights chosen. The order of the nodes in the priority queue also determines the left and right child nodes of the new root. The newly created tree (or its root node) is inserted into the priority queue. We now have a forest of seven trees as shown here. Although the newly created node has the same weight as `Space`, it is ordered after Space in the priority queue because `Space` is an ASCII character. **In PE09 and PA05, we will use the ordering strategy as outlined here.**



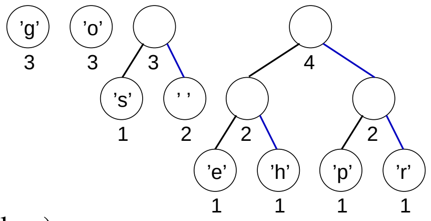
Removing the first two (minimal) nodes in the priority queue yields another new tree with weight 2 as shown here. There are now six trees in the forest of trees that will eventually build an encoding tree.



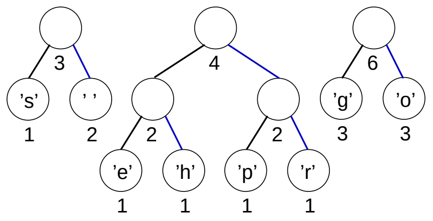
Again we must remove the first two (minimal) nodes in the priority queue. The lowest weight is the 's'-node/tree with weight equal to 1. There are three trees with weight 2; the one chosen corresponds to an ASCII character because of the way we order the nodes in the priority queue. The new tree has a weight of 3, which will be placed last in the priority queue according to our ordering strategy.

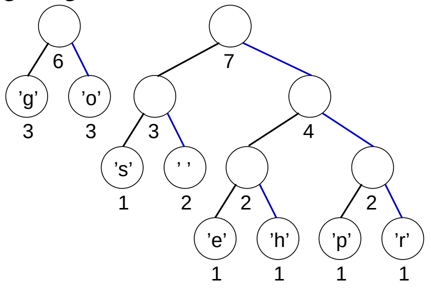


Now there are two trees with weight equal to 2. These are joined into a new tree whose weight is 4. There are four trees left, one whose weight is 4 and three with a weight of 3.



The first two minimal (weight 3) trees in the priority queue are joined into a tree whose weight is 6 (see the left figure below). There are three trees remaining. Now, the minimal trees have weights of 3 and 4; these are joined into a tree with weight 7 (see the right figure below).





Finally, the last two trees are joined into a final tree whose weight is 13, the sum of the two weights 6 and 7 (see Section 2.3, page 2).

Note that you can easily come up with an alternative Huffman coding tree by using a different ordering strategy to order trees of the same weights. In that case, the bit patterns for each character are different, but the total number of bits used to encode `"go go gophers"` is the same.

2.5 Huffman coding or compression

In PE09 and PA05, you will implement a subset of functions that are required in Huffman coding and decoding. Although you will not implement a full-scale Huffman coding and decoding, we will still present the basic steps in Huffman coding and decoding so that you have a complete picture.

To compress a file (sequence of characters), we are required to have a table of bit encodings, i.e., a table giving a sequence of bits used to encode each character. This table is constructed from a coding tree using root-to-leaf paths to generate the bit sequence that encodes each character. A compressed file is obtained using the following top-level steps:

1. Build a Huffman coding tree based on the number of occurrences of each ASCII character in the file.
2. Build a table of Huffman codes for all ASCII characters that appear in the file.
3. Read the file to be compressed (the plain file) and process one character at a time. To process each character find the bit sequence that encodes the character using the table built in Step 2 and write this bit sequence to the compressed file.

The main challenge here is that when you encode an ASCII character read from the file, the Huffman code is typically shorter than 8 bits. However, most systems allow you to write to a file a byte or 8 bits at a time. It becomes necessary for you to accumulate the Huffman codes for a few ASCII characters before you write an 8-bit byte to the output file. (You will get to do this in PA05.)

To compress the character stream `"go go gophers"` (without `'\0'`) for example, we read from the character stream one character at a time. The Huffman code for `'g'` is `00` (**for Huffman code, left to right is least significant to most significant**), we cannot write to the file yet. We read the next character `'o'`, whose Huffman code is `01`. Again, we cannot write to the output file because the total number of bits is only 4. Now, we read the next character `Space`, whose Huffman code is `101`. We have now accumulated 7 bits. Now, we read `'g'` again. The least significant bit of the Huffman code of `'g'` (`0`) is used to complete the byte, and we can now print a byte of bit pattern `01011000` (**in the conventional notation that the left most bit is most significant**) to the output file.

The most significant bit of the Huffman code of `'g'` (`0`) is the least significant bit of the next byte in the file. This byte will also contain the bits of the next three characters `'o'` (`01`), `Space` (`101`), and `'g'` (`00`), and the 8-bit pattern written to the file is `00101100`.

The following byte contains bits of `'o'` (`01`) and `'p'` (`1110`), and the two least significant bits of `'h'` (`11`) for a bit pattern of `11011110`. The two most significant bits of `'h'` (`01`), the bits of `'e'` (`1100`), and the two least significant bits of `'r'` (`11`) form the next byte of bit pattern `11001110`.

The last byte in the file contains only 5 useful bits: the two most significant bits of `'r'` (`11`) are in the least significant positions of this byte and the 3 bits of `'s'` (`100`). As unused bits, the 3 most significant bits of the last byte are 0. The bit pattern is `00000111`.

2.5.1 Topology Information

The compression program must store some information in the compressed file that will be used by the decompression (decoding) program.

To store the tree in a file, we use a `pre-order traversal`, writing each node visited as follows: when you encounter a leaf node, you write a 1 followed by the ASCII character of the leaf node. When you encounter a non-leaf node, you write a 0.

For our example, the topology information can be stored as `"001g1o001s1 001e1h01p1r"`, based on pre-order traversal. In this example, we use characters `'0'` and `'1'` to distinguish between non-leaf and leaf nodes, respectively. As there are eight leaf nodes, there are eight `'1'` characters and there are seven `'0'` characters for non-leaf nodes. This approach used a total of 23 bytes. We refer to this as a character-based representation of the Huffman coding tree.

The representation of a Huffman coding tree can be made more economical if we use bits 0 and 1 to distinguish between non-leaf and leaf nodes. In this example, there will be a total of 79 bits (64 bits for the ASCII codes of the eight leaf nodes, 8 1-bits for the leaf nodes, and 7 0-bits for the non-leaf nodes).

For example, in the bit-based approach, the first 16 bits (or the first 2 bytes **in the convention that left to right is MSB to LSB**) describing the topology of the Huffman tree constructed for encoding the character stream `"go go gophers"` are `00111100 11111011`.

The least significant two bits of the first bytes are 0-bits (for non-leaf nodes). The next bit is a 1-bit (for a leaf node). It is followed by the ASCII code for `'g'`, which has a bit pattern of `01100111`. The 5 least significant bits (`00111`) are in this byte. The 3 most significant bits (`011`) are in the least significant position of the next byte. This is followed by another 1-bit, followed by `1111`, the 4 least significant bits of the ASCII code for `'o'`, which has a bit pattern of `01101111`.

Note that ASCII code for 'g' straddles two bytes. Similarly, the ASCII code for 'o' straddles two bytes.

In the bit-based representation of the Huffman coding tree, the last byte may not contain 8 bits. In this case, the most significant unused bits of the last byte should be assigned 0.

In PE09, you focus on the construction of a Huffman coding tree and the storage of a Huffman coding tree in a file using the character-based representation. You also deal with the building a table of Huffman codes (Step 2 in Section 2.5). In PA05, one of the tasks is to deal with the storage of a Huffman coding tree in a file using the bit-based representation.

2.6 Decoding or decompression

Given a compressed (or coded) file, which contains, among other things, the topology information and a bit stream corresponding to the encoding of the original file, the decompression program should perform the following tasks:

1. Re-build a Huffman coding tree based on the topology information.

2. Decode the bit stream to obtain the original character string/file. We must initially start from the root node of the Huffman coding tree. When we are at a leaf node, we print the corresponding ASCII character to the output file and re-start from the root node again. When we are at a non-leaf node, we have to use a bit from the bit stream in the compressed file to decide how to traverse the Huffman coding tree (0 for left and 1 for right).

In PA05, you will work on the (re-)building of a Huffman coding tree based on the topology information stored in the file. We will not perform the actual decoding.

3 Program you have to write

For this exercise, you are given the flexibility of designing your `.h` file(s) and and `.c` files. The compilation of your source files must generate an executable, which is called `pe09`. The executable takes in an input file, and produces four output files:

```
./pe09 input_file output_file_1 output_file_2 output_file_3 output_file_4
```

The input file (`input_file`) can be any files of size up to $2^{(63)} - 1$ (`LONG_MAX`) characters. It is acceptable for the input file to be empty. Some small sample input files are given in the `examples` folder: `gophers`, `stone`, `empty`, and `binary1`.

The four output files (`output_file_1`, `output_file_2`, `output_file_3`, and `output_file_4`) are as follows:

1. `output_file_1`: A "binary" file with 256 `long` integers as output. For $0 \leq i < 256$, the ith `long` integer should be the number of occurrences of the ith ASCII character in the input file. The size of each file is $256 \times$ `sizeof(LONG)`.

   The corresponding output files for the sample input files are `gophers.count`, `stone.count`, `empty.count` and `binary1.count` in the `examples` folder.

2. `output_file_2`: A file with only 1 line if the newline character is not present in the input file. Otherwise, there should be 2 lines in this file. This is a file that shows only characters that have appeared in the input file. Moreover, the characters will appear in ascending order with respect to the numbers of occurrences. If two characters have the same number of occurrences, the character with the higher ASCII value should appear later. The format of the file is shown as a linked list:

   ```
   character_1:count_1->character_2:count_2->...->character_k:count_k->NULL
   ```

   where `character_1` through `character_k` are ASCII characters that appear in the input file. `count_1` $\leq$ `count_2` $\leq$ ... $\leq$ `count_k`. If `count_i` is the same as `count_i+1`, `character_i` $<$ `character_i+1`. There is a newline after `NULL`. The `character_i:count_i` pair is printed using the format string `"%c:%ld"`.

   The easiest way to get this output file is to enqueue the characters (together with their counts) into a priority queue, and then print the list (see PE08). You should design and define a structure that would allow you to enqueue a character and its count at the same time into a priority queue. Take into consideration what you have to do for the third output file and fourth output file before you decide on your user-defined structure.

   The corresponding output files for the input files in the `examples` folder are `gophers.sorted`, `stone.sorted`, `empty.sorted`, and `binary1.sorted`.

3. `output_file_3`: This file should store the character-based representation of the Huffman coding tree you have constructed. If there are $n > 0$ distinct ASCII characters in the input file, there should be exactly $3n - 1$ characters in the corresponding file in the folder tree. See Section 2.5.1 for the description of character-based representation. Note that in this character-based representation, the topology information is printed in pre-order fashion. For an empty coding tree, the corresponding file to store its character-based representation should be an empty file. The following output files for the input files in the examples folder are `gophers.tree`, `stone.tree`, `empty.tree`, and `binary1.tree`.

4. `output_file_4`: The file contains as many lines as the number of distinct characters that appear in the input file. Each distinct character should have a line in this output file with the format of

   ```
   character:stream_of_0_and_1
   ```

where the stream of `0` and `1` (as character `'0'` and character `'1'`) being the binary pattern corresponding to the Huffman code of the character. The following output files for the input files in the examples folder are `gophers.code`, `stone.code`, `empty.code`, and `binary1.code`.

To produce this output file, it is essential that you build a Huffman coding tree based on the description in Section 2.4. The construction of a Huffman coding tree requires you to maintain a priority queue of partially constructed Huffman coding trees. In these coding trees, all leaf nodes are ASCII characters with their respective counts in the input file.

Therefore, we suggest that you define structure(s) that could be used for the generation of all output files. In particular, a tree structure with fields in each node to store an ASCII character and its count would be useful. While a leaf node will store an ASCII character, you will have to decide what should be stored in the same field for a non-leaf node. The count field of a node should store the total of the counts in the leaf nodes below the node.

As mentioned in Section 2.4, the priority is defined based on the counts. If two trees have the same total count (of their respective leaf nodes), a tree that corresponds to a single leaf node should have a higher priority. If both trees are leaf nodes themselves, the ASCII (numeric) values should determine the priority; a leaf node with a lower ASCII (numeric) value is ordered first in the priority queue. Otherwise, the order in which the trees are created should determine the priority; a tree created early is ordered first in the priority queue. When two trees are dequeued from the priority queue, the first node dequeued should be the left branch of the newly constructed coding tree, and the second node dequeued should be the right branch of the newly constructed coding tree. You have a Huffman coding tree when there is only tree in the entire list.

Note that we impose this priority scheme for ease of grading. In practice, as long as we use the total counts of occurrences of characters in the respective trees in the priority queue to perform the enqueuing and dequeuing operations, the resulting tree is a Huffman coding tree. In other words, there could be multiple Huffman coding trees for a single input file. However, based on the priority scheme defined in this exercise, there is only one Huffman coding tree that fits the bill.

The order in which you print the characters and their codes in the third output file is determined by a tree traversal that visits the left branch followed by the right branch. In the tree traversal process, a left branch correspond to a `'0'` and a right branch corresponds to a `'1'`. (In reality, it is a 0-bit and a 1-bit, but for the purpose of this exercise, we use `'0'` character and `'1'` character in the fourth output file). You print the character and its code when you reach a leaf node in the tree traversal.

If the input file cannot be opened, the program should not produce any output files and it should return `EXIT_FAILURE`. If the input file can be opened for reading, the corresponding output files should be produced. If all output files can be produced, the program should return `EXIT_SUCCESS`. Otherwise, the program should return `EXIT_FAILURE` at the first failure. For example, if you could not produce the third output file (because of, for example, failure in memory allocation or failure in opening the output file), you should `EXIT_FAILURE` immediately. In such a case, the program should have produced only the first and second output files.

Regardless of the exit status of the program, the program should not have any memory issues (as reported by `valgrind`).

4 Compiling your program

As you have advanced to this stage, we believe that you have the necessary experience to decide what `gcc` command should be used in your `Makefile`. We will use the command “`make pe09`” to create an executable named `pe09`.

5 Submission

You must submit a zip file called `PE09.zip`, which may contain any number of files. You must also include in the zip file a `Makefile` that allows us to generate an executable called `pe09` by using the command “`make pe09`.”

Assuming that you are in the folder `PE09`, which contains all your source files (`.c` and `.h`) and the `Makefile`, use the following command to zip your files:

```
zip PE09.zip *.[ch] Makefile
```

Make sure that you name the zip file as `PE09.zip`. Moreover, the zip file should not contain any folders. Submit `PE09.zip` through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

6 Grading

You should submit only `.c` files, `.h` files, and a `Makefile` in the zip file `PE09.zip`. We will use the `Makefile` to compile and we expect an executable called `pe09`. The command `make pe09` should generate `pe09`. If we fail to obtain `pe09`, you do not receive any credit for this exercise.

As mentioned earlier, the executable `pe09` expects 5 arguments: 1 input filename followed by 4 output filenames.

You earn points by producing the correct output files. The first output file accounts for 20%, the second output file 20% points, the third output file 60% points. The first three output files account for the full credit of the exercise. Up to 5 points will be deducted if your `main` function does not behave as described in Section 3.

The fourth output file allows you to earn 100 bonus points (which count towards the PE component of the overall grade).

Even if you are not producing the fourth output file because you are not interested in the bonus points, you should still write the `main` function assuming that the fourth output file(name) will be provided as one of the arguments to the `main` function. If you are able to produce the first three output files properly, your `main` function should assume that you could produce the fourth output file properly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

7 A few points to remember

All debugging messages should be printed to `stderr`. If the program creates an output file when it should not, or if the output file is incorrect, you do not receive credit for that output file.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.