Assignment 5

CPSC 457 Fall 2022

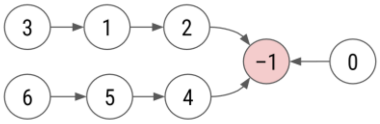
Due date is posted on D2L.
Individual assignment. Group work is NOT allowed.
Weight: 8% of the final grade.

File Allocation Table Checker

For this assignment you will write a function that examines the contents of a File Allocation Table (FAT). Your function will determine the largest possible block chain(s) within the FAT and then return a list of all blocks where such chains could start. Your code will need to be able to handle both normal and corrupted FATs. Normal FATs do not contain cycles, nor shared blocks, while corrupted FATs can contain shared blocks and/or cycles.

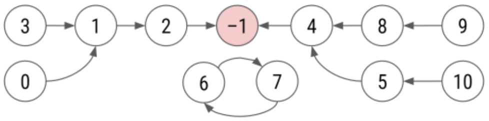
Below are two examples of FATs; the left one is not corrupted, the right one is corrupted. Please note that we are using (-1) to denote a NULL pointer.

FAT=[-1,2,-1,1,-1,4,5]



Largest chain has length 3. Possible starting locations are blocks 3 or 6.

FAT=[1,2,-1,1,-1,4,7,6,4,8,5]



Largest chain has length 3. Possible starting locations are blocks 0, 3, 9 or 10.

In both FATs above, the longest possible chain has length 3. The FAT on the left has 2 such chains, one starting on block 3, and the other starting on block 6. In the FAT on the right, there are 4 possible starting locations for the longest chain: blocks 0, 3, 9 and 10.

Starter code

Start by downloading and compiling the starter code:

```
$ git clone https://gitlab.com/cpsc457f22/fatsim.git
$ cd fatsim
$ make
```

Only modify and submit the file `fatsim.cpp` by implementing the `fat_check()` function. Do not modify any other files.

The included driver code will parse the FAT contents from standard input and then call the `fat_check()` function. The driver expects the standard input to contain integers separated by white space. After `fat_check()` returns, the driver will display the results.

Inputs and outputs

The function `fat_check()` has the following signature:

```
std::vector<long> fat_check(const std::vector<long> & fat);
```

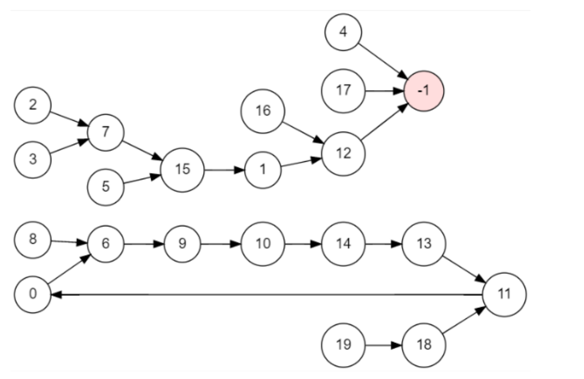
The `fat` parameter will contain the entries of the file allocation table. The meaning of these integers will be the same as we discussed during lectures, i.e. `fat[b]` represents block `b`'s next pointer. Every entry in `fat` will be in the range $[-1, N)$, where a value of (-1) will represent a NULL pointer (an end of chain).

The `fat_check()` function must return all block numbers where the longest chain of blocks could start, sorted in ascending order.

Below is a sample FAT test file, the expected output, and a graphical representation of the FAT:

```
$ cat tests/fat3.txt
  6 12  7  7 -1 15  9 15  6 10
 14  0 -1 11 13  1 12 -1 11 18

$ ./fatsim < tests/fat3.txt
blocks: 2 3
elapsed time: 0.00s
```



For example, the first integer '6' in the above FAT represents the fact that the next pointer of block '0' is block '6'. The longest chain has length 5, and starts either on block 2 or 3. Therefore, for the above input, the `fat_check()` function should return `[2,3]`.

Limits:

The number of entries in FAT will be in the range [1…10,000,000].

Marking

- Your code will be marked for correctness and efficiency.
- Your mark will be based on the number of tests your solution will pass.
- To get full marks, you will need to implement an $O(n)$ solution, so that it can finish on inputs with 10 million FAT entries under 10s, or inputs with 1 million entries under 1s.
- For partial marks (around 60%), your solution will need to be able to finish under 10s for any inputs with up to 40,000 FAT entries.
- Small number of test inputs are provided for you in the `tests` subdirectory, but you should create your own test inputs as well.

Submission

Submit your `fatsim.cpp` file to D2L.

General information about all assignments

1. All assignments are due on the date listed on D2L. Late submissions will not be marked.
2. Extensions may be granted only by the course instructor.
3. After you submit your work to D2L, verify your submission by re-downloading it.
4. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
5. Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
6. All programs you submit must run on `linuxlab.cpsc.ucalgary.ca`. If your TA is unable to run your code on the Linux machines, you will receive 0 marks for the relevant question.
7. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on linuxlab.cpsc.ucalgary.ca, when compiled with `-O2` optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
8. **Assignments must reflect individual work**. Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (including code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly. This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
9. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix - hints

No advanced data structures are needed at all.

A naive solution would test the chain length by starting from every block and counting how many times you need to follow the next pointer to reach “-1”, while, of course, avoiding getting into an infinite loop due to cycles. This would lead to an $O(n^2)$ solution.

The most efficient solution would run in $O(n)$ time. Such solution is quite easy to achieve as well, without using any advanced data structures. For example, consider creating node adjacency lists, like I suggested for the previous assignment. Adjacency list can be easily implemented using `std::vector<std::vector<long>>` type. Once you have an adjacency list, it is easy to figure out maximum chain lengths if you start tracing the graph from the ‘-1’ node, for example using DFS traversal.

The starter code contains a python solution called `fatsim.py`. This is an **inefficient**, $O(n^2)$ solution, and it will only work for small inputs. You can use this python script to test your own code and to design your own test files.

Since most students have limited file system quotas, I made a FAT generator called `rngfat.py`. It takes 2 command line arguments: size of the FAT table, and an RNG seed. It will always generate the same output for the same command line arguments. You can use it like this:

```
$ ./rngfat.py 100 10 | ./fatsim
FAT has 100 entries.
blocks: 12 78 84
elapsed time: 0.00s
```

A medium sized input:

```
$ ./rngfat.py 100000 10 | ./fatsim
FAT has 100000 entries.
blocks: 41643 53888 93180
elapsed time: 0.01s
```

A very large input:

```
$ ./rngfat.py 9999999 10 | ./fatsim
FAT has 9999999 entries.
blocks: 4377769 7165176
elapsed time: 3.07s
```