Smart File Storage

Traditional file storage systems keep files contiguously on a storage device "as-is." This means that the file is stored without modification on the disk: every bit and byte in the content is kept and stored in order.

This is great for simplicity, but in certain scenarios this can be extremely inefficient. Two of which are:

1. Having the same file stored in different folders. In a traditional file system, this would lead to the information getting copied for each instance of the file in a different folder. If the file were N bytes long and it was copied in 5 different folders, the total storage space (ignoring file system overhead) would require 5*N bytes.

2. Having information or content in a file that is repeated, leading to redundant information stored in the disk. An example would be a file that has the same 10 bytes repeated 100000 times, requiring 1MB of storage instead of 10 bytes (ignoring file system overhead).

A "smart file storage" system would recognize these potential inefficiencies and work out solutions to more efficiently use the disk space.

For the purpose of this challenge, we will only focus on the 2nd inefficiency (redundancy in a file).

Challenge: Create a program that (in memory or on disk) splits up a binary file in to "blocks" of a given size, and outputs the minimum disk space (in bytes) required to store the information in a smart file storage system (ignoring file system overhead). To do so, you will need to detect repeated content (i.e., repeated blocks) in a file. If the file described in bullet point (2) above was given as the input and a block size of 10 was specified, the answer outputted would be 10 (a minimum of 10 bytes can be used to store the file, ignoring file system overhead).

Your program (in any language) takes exactly 2 parameters on the command line separated by a single white space:

- <filename> -- the path to a local file to use as input.
- <block_size> -- the size of blocks (pieces) that a file should be split in to, given in bytes.

Usage**:** ./deconstruct <filename> <block_size>

Output: A single integer the represents the minimum number of bytes the file can be stored on the disk as non-duplicated blocks (ignoring file system overhead). This should be equal to the number of non-duplicate blocks times the block size (i.e., block_size * non_duplicate).

Bonus: Instead of only outputting the minimum number of bytes that the file can be stored as, have your program break the input file in to blocks and only store non-duplicate blocks on the disk. Therefore, the total file space size of the non-duplicate blocks outputted should be equal to the minimum number of bytes your program calculated. Then, create a second program "./reconstruct" that will intelligently reconstruct the original file from the non-duplicated blocks that you are now storing on the disk.

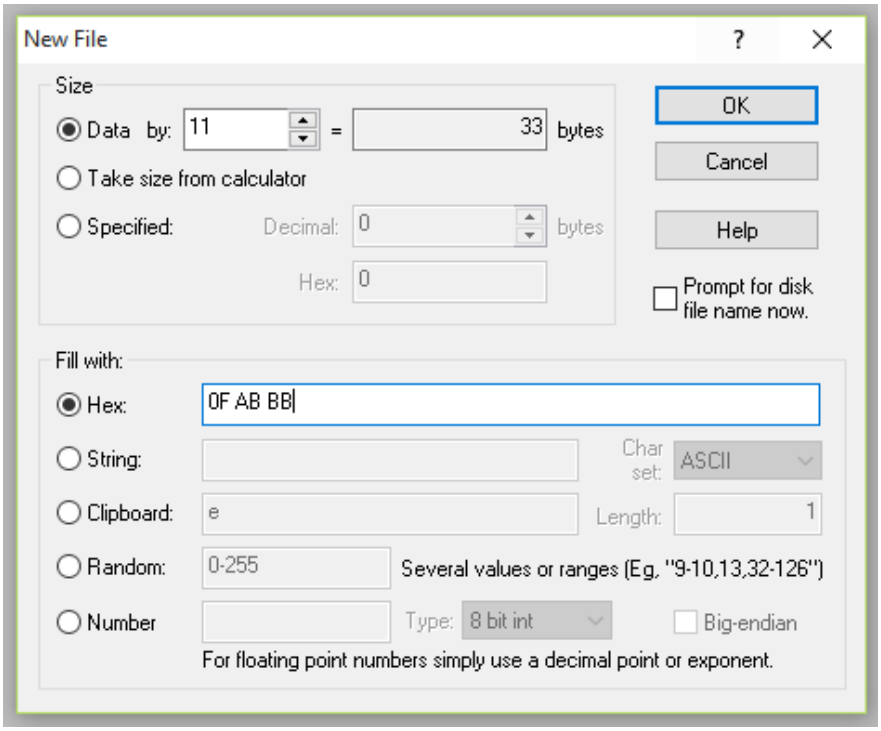
Hint: Your "deconstruct" program may need to create meta-data (i.e., "file system information") that "reconstruct" can use to re-build the original file from the non-duplicate blocks you stored on the disk.

FAQ

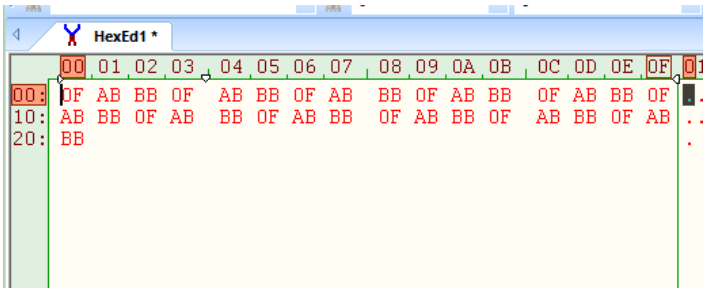
Q: Do blocks need to be created exactly from byte 0, or can I optimize de-duplication by using different offsets?
A: Treat blocks as non-sliding and drawn at exact boundaries and the block size cannot change. It's fixed and it's one of your inputs. So, if your block size is 4 then bytes 0-3 belong to block 1, bytes 4-7 belong to block 2, bytes 8-11 belong to block 3, etc

Q: Do you have any test files that I can use?
A: We do not provide any test files, but you can easily generate some using the tool HexEdit.

You can specify what to "fill the file" with (e.g., I filled with 0F AB BB), and I filled it 11 times for a total of 33 bytes:



After clicking "OK" I see a binary file with 0F AB BB repeated 11 times, for a total of 33 bytes:



You can then save it to your disk somewhere.