Exercise 1 (1/4)

- You are tasked with implementing a simple memory allocator simulation in C.
- Write a program **allocator.c** which creates an unsigned **integer** array for storing 10^7 integers (maximum memory size) to serve as memory cells. On startup, all cells are initialized to zeros (0) (i.e. all memory cells are free).
- Implement the following functions:
  - `void allocate_<algo>(adrs, size)`:
    - This function should find a contiguous free cells from the array for storing at least n integers where $n =$`size`, then it should set these cells to the value `adrs`.
    - `adrs` will act as an identifier for reserved cells. It should be a non-zero unsigned integer and does not represent an index in the array since the indices of allocated cells should be determined by the allocator at run-time and based on the selected algorithm and status of the memory.
    - `size` can take values between 1 and 10^7.

Exercise 1 (2/4)

- `void clear(adrs)`:
  - This function searches for memory cells that hold the given address (`adrs`) and frees it.
  - `adrs` should be a non-zero unsigned integer.

- **Note: for simplicity, we consider here that `adrs` are only identifiers for the allocated memory cells which can be used by `clear` queries.**
- You are required to implement each of the following: First Fit, Best Fit, Worst Fit, to use as the algorithm for the allocate function. At the end, you should have 3 allocate functions, one function for each algorithm.
- Your program should accept input from a file named `queries.txt`. The file will contain a list of queries, with each query on a new line. The end of the file will be indicated by the line "end".

Exercise 1 (3/4)

- The queries in the file can be one of two types:
  - `allocate adrs size`: Allocate memory of the specified `size` for the given `adrs`.
  - `clear adrs`: Clear the memory space associated with the given `adrs`.
- **Note:** we assume that the file **queries.txt** always contains valid input (e.g. there is no clear query for an address not allocated). You just need to check if the file exists.
- For each of the three allocation algorithms, you should reset the memory array (free all memory cells), execute all the queries from `queries.txt` file, and then print the throughput (performance metric). Throughput in this context simply refers to the number of queries executed divided by the total allocation time.

Exercise 1 (4/4)

- Example for `queries.txt`:

```
allocate 1001 512
allocate 2002 256
clear 1001
end
```

- You can find a bigger sample for `queries.txt` from here.
- You should submit **allocator.c**. You also need to compare the performance of the algorithms (assuming that they are receiving the same input queries) and add the performance results and your findings to **ex1.txt**. Submit **ex1.txt** too.
- The total allocation time starts from the very first query and ends after executing the last query. You can use the function **clock()** from **time.h**.

Exercise 2 (1/2)

- In this exercise, you should use `mmap` system call to a big text file **text.txt** in chunks.
- This exercise requires generating a relatively large file of size *500MiB*.
- The content of the text file **text.txt** should be generated from "/dev/random". The file "/dev/random" is a special character file that generates pseudorandom numbers.
- Write the program **ex2.c** to perform the following tasks using memory mapping.
  - Create the empty file **text.txt**. Open the file "/dev/random", and read a character c at a time.
  - If the generated character c is a printable character (use **isprint** function from **ctype.h** to check it), then we should add it to the file **text.txt**. Otherwise, we ignore the character and generate a new character.

Exercise 2 (2/2)

- You need to add a new line to **text.txt** after adding 1024 characters (max line length). You should continue adding characters till you get a file of size *500 MiB= 500*1024KiB*.
- After finishing the process above, you would get a relatively large file in your file system.
- Open the whole file **text.txt** in chunks where the chunk size is *1024th multiple of the page size in your system*. You can get the page size in C as follows: (if page size is *4KiB* then chunk size is 4MiB)

```
#include <unistd.h>
long sz = sysconf(_SC_PAGESIZE);
```

- Count the capital letters in the mapped chuncks. Print the total number of the capital letters in the file to `stdout`.
- Replace the capital letters with lowercase letters in the file.
- `unmap` the mapped memory.

- Submit `ex2.c`