1. 開發環境：Windows10 + VScode + TDM-GCC + C++2a
2. 資料結構：

   struct Data; // 用於儲存針對單一 frame 的資料(frame, queue, fault)

   struct SUM_UP; // 用於儲存單一方法的最後一行的統計訊息

   對於每一個 page 中的對應之 queue 使用 std::vector 儲存

   對於 LFU、MFU 的計數使用 std::unordered_map 實現類似 hash map 的功能
3. 運作流程：

   讀檔 -> 根據 method 執行對應的 function(s) -> 寫檔 -> 結束
4. 實作方法：

   ※所有方法都是依序 iterate 單一 frame(page)

   甲、FIFO：

   (a) 當 queue 沒有目前的 page 時：

   若 queue 已滿，把 queue 中最後一個作為 victim 移出，更新 page replace 次數

   更新 page fault，並在 queue 最前面插入目前的 page

   (b) 將 queue 轉為 std::string 以便寫檔

   乙、LRU：

   (a) 當 queue 沒有目前的 page 時：

   若 queue 已滿，把 queue 中最後一個作為 victim 移出，更新 page replace 次數

   更新 page fault，並在 queue 最前面插入目前的 page

   (b) 否則就刪除 queue 中已經存在的、與目前相同的 page，並在 queue 最前面插入目前的 page

   (c) 將 queue 轉為 std::string 以便寫檔

   丙、LFU + FIFO：

   (a) 更新對應目前 page 的 count

   (b) 當 queue 沒有目前的 page 時：

   若 queue 已滿，先找到 queue 中最少的 count，再反向找到對應的第一個 page 作為 victim，將其 count 歸零，移出 queue，更新 replace 次數

   更新 page fault，並在 queue 最前面插入目前的 page

   (c) 更新 page fault，並在 queue 最前面插入目前的 page

丁、MFU + FIFO：

(a) 更新對應目前 page 的 count

(b) 當 queue 沒有目前的 page 時：

若 queue 已滿，先找到 queue 中最多的 count，再反向找到對應的第一個 page 作為 victim，將其 count 歸零，移出 queue，更新 replace 次數

(c) 更新 page fault，並在 queue 最前面插入目前的 page

戊、LFU + LRU：

(a) 更新對應目前 page 的 count

(b) 當 queue 沒有目前的 page 時：

若 queue 已滿，先找到 queue 中最少的 count，再反向找到對應的第一個 page 作為 victim，將其 count 歸零，移出 queue，更新 replace 次數

(c) 否則就刪除 queue 中已經存在的、與目前相同的 page，並在 queue 最前面插入目前的 page

(d) 更新 page fault，並在 queue 最前面插入目前的 page

己、All：依上面方法順序全部執行一遍

5. 分析不同方法之次數比較：

Method	*Page fault (input1, input2)*	*Page replace (input1, input2)*
FIFO	(9, 15)	(6, 12)
LRU	(10, 12)	(7, 9)
LFU + FIFO	(10,13)	(7, 10)
MFU + FIFO	(9, 15)	(6, 12)
LRU + LFU	(10, 11)	(7, 8)

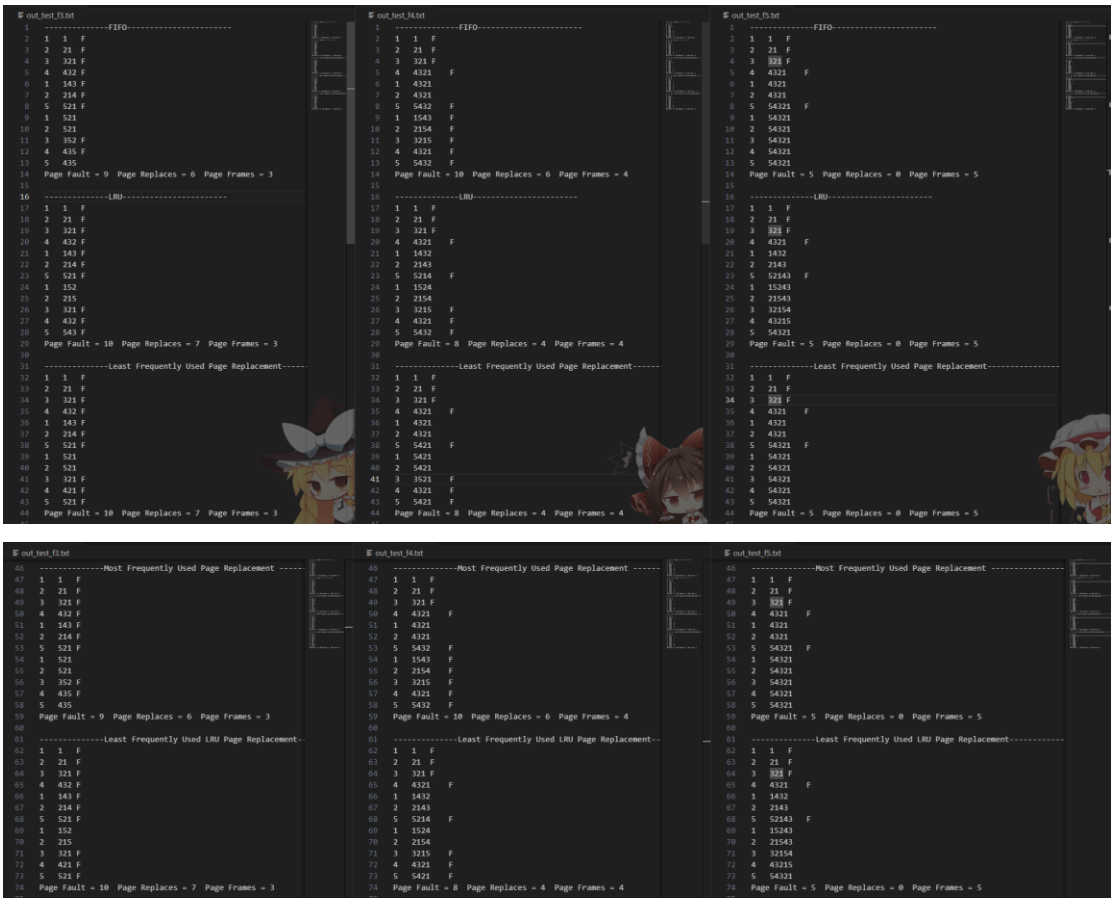
6. 結果與討論：

1. 實驗數據與發現：

- 若用(input1 + input2)/2 由小到大排序 fault 與 replace 都是：
  LRU + LFU < LRU < LFU + FIFO < FIFO = MFU + FIFO
- 可得出 FIFO 與 MFU + FIFO 最差，LRU + LFU 最佳
- 經常 access 的 page 是不應該做為 victim 的，所以 MFU + FIFO 表現最差
- FIFO 與 MFU + FIFO 具備相同的表現，但程式碼與時間複雜度較低，若要實現 MFU + FIFO 的效果不如直接使用 FIFO
- LRU + LFU 雖然表現最佳，但程式碼與時間複雜度也最高，消耗大

2. 畢雷迪反例：

   在 page frames = 3, 4, 5 的限制之下，1234512512345 之表現：



- 觀察到 FIFO 與 MFU + FIFO 在 page frames = 3, 4 時 page fault 次數由 9 增加至 10，且 page replace 次數沒有減少，可得知這兩個表現最差的演算法會有機率導致 Belady's Anomaly 發生
- 由在 page frames = 5 時發現此兩種演算法並沒有繼續增加 page fault 次數，page replace 次數也有減少，可得知 Belady's Anomaly 為特例情況