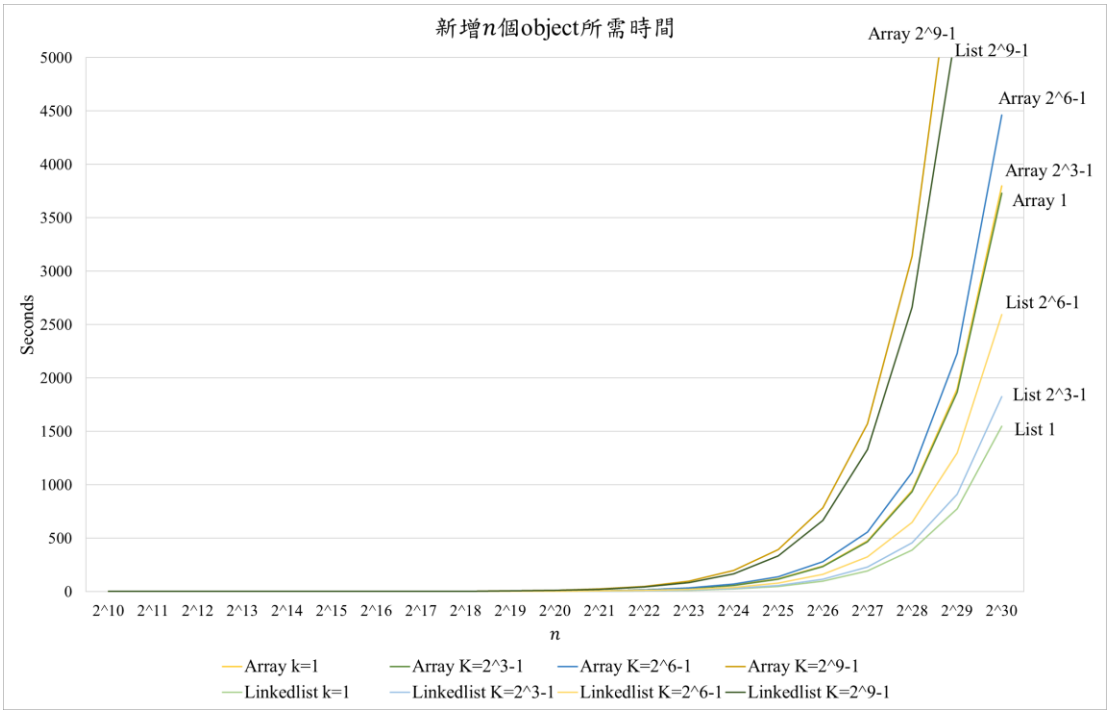
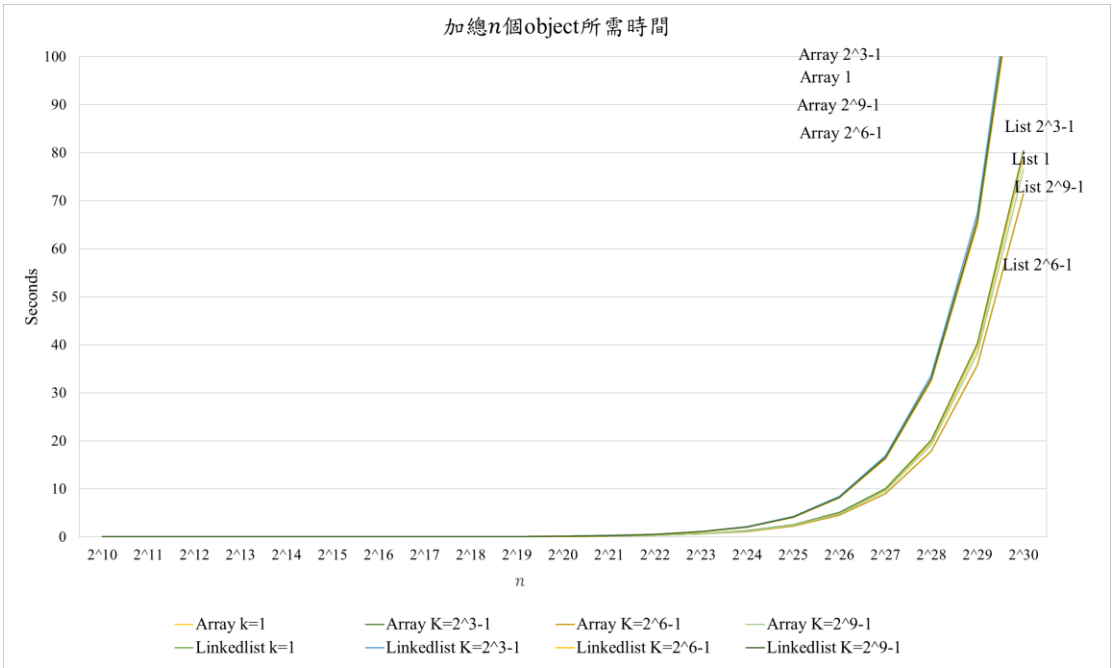
實作目標：比較dynamic array與linked list，並討論cache的影響。
Object定義：dynamic array與linked list會儲存n個object，每個object包含一個int num與一個int foo[k]，其中k=1, 2^3-1, 2^6-1, 2^9-1。
Object產生方式：num由0~9999的整數中隨機選擇，foo的內容不需特別指定。
比較方法：

- 對於每個k值設定，測量上述資料結構新增$n = 2^h$個object所需時間，其中h = 10, 11, 12, …, 30。
- 新增n個object後，計算這n個object中的num總和，並測量所需時間。（測量的時間不包含新增object所需時間）
- 為了降低誤差，重複上述實驗10次後取平均。

一、折線圖與解釋實驗結果



- 假說一：linked list 較快，只要找到前一個值的 pointer，就可以接到新增的數字上，但 array 如果新增在後面幾乎所有格子都要動，另外，dynamic array 原位置的 array 滿時要新開 array，O (n)複製貼上到新 array，也會比較慢。
- 實驗結果和假說相同。
- 假說二： foo[k]越大，新增 object 再加入 linked list、array 的時間越久。
- 實驗結果和假說相同，foo[k]越大，需要新增加入兩種資料結構的時間越久。



- 假說一：dynamic array 加總較快， CPU 和 RAM 要資料時，RAM 會把附近資料一起存到 cache，dynamic array 因為是連續型儲存資料，CPU 要計算時，大多附近的資料已經存到 cache，可以直接和 cache 拿不用再到 RAM，所以計算較快，但 linked list 是離散型，取附近資料可能不是下一個節點，所以計算時常需要從 CPU 拿。
- 實驗結果和假說相同。
- 假說二： foo[k]越大，cache 越沒有空間放 num（cache 使用效率變差），CPU 需要到 RAM 拿資料，加總會變慢
- 實驗結果和假說不同，foo[k]對加總時間影響不明顯，甚至有倒置狀況。

二、心得

- 前兩周為了測量到 n = 2^25 以上的時間，追求能夠用最省空間的寫法，把 foo 嘗試寫成儲存個數（如 1、7、63、511，4 個數字），而不是實際數量，結果造成 linked list 和 dynamic array 內部比較花費時間時，k 的數量不影響執行時間。
- 追求能夠用最省空間，進而把生成 object 寫在迴圈外，使得要生成 2^10 個 object 時，都是迴圈到同樣的 num，全部的 object 數字相同，可能影響加總時間

- 有趣的發現是，原先測試一直 array 遍歷加總較慢，跟假設 array 可以更有效利用快取的假設不同，原本以為是寫法不同適用第二個版本，結果主因是沒有運用 array[index]的功能，使 array 一樣要遍歷所有 object 的 num 屬性。
- Linked list 的迴圈需要另外加成員函式。