**演算法程式作業三**

110403518 林晉宇

1. **fur­thest-in-future**

本次作業題目為furthest-in-future管理cache的演算法，並透過C++實作來模擬cache的運作。

報告包含:

1. 實作程式的pseudo code以及資料結構的說明
2. 複雜度分析
3. 如何透過python生成測資
4. 程式執行時間圖
5. **實作方法及資料結構說明**

我參考楊弘笠同學在作業7-1簡報中的作法:

1. 建一個**record**陣列，用來記錄下一個相同request的index。
2. 每次讀入新的request時，判斷是否在cache [1]:
3. 在cache，則cache hit。
4. 不在cache，且cache還有空間，則cache miss，並把request值加入cache。
5. 不再cache且cache已滿，則cache miss，並選出在request sequence中最遠的元素並移除 [2]。

* 資料結構說明:

**[1]** 使用C++的**unordered\_map (hash map)**，增刪查找為近乎常數時間。

**[2]** 使用C++的**priority\_queue<pair<int,int>>**，pair中第一個值代表下一個相同request的index(即record陣列所存的值)，第二個代表該request的值。priority\_queue目的在於確保距離最遠的request會在priority\_queue的top()，其在新增及刪除元素為**O(logn)**。

* **Pseudo code:**

    unordered\_map<int,bool> cache;

    int cur\_cache\_size = 0; // 當前cache有多少元素

    priority\_queue<pii> pq;

    for(i = 1 ~ n) // 1-based

    {

        if(cache[request[i]]) // cache hit

            pq.push({record[i], request[i]});

        else if(cur\_cache\_size < k)

        {

            cur\_cache\_size++;

            cache[request[i]] = true;

            pq.push({record[i], request[i]});

        }

        else

        {

            int ev = pq.top().second; // 要evict的request值

            cache[request[i]] = true;

            pq.push({record[i], request[i]});

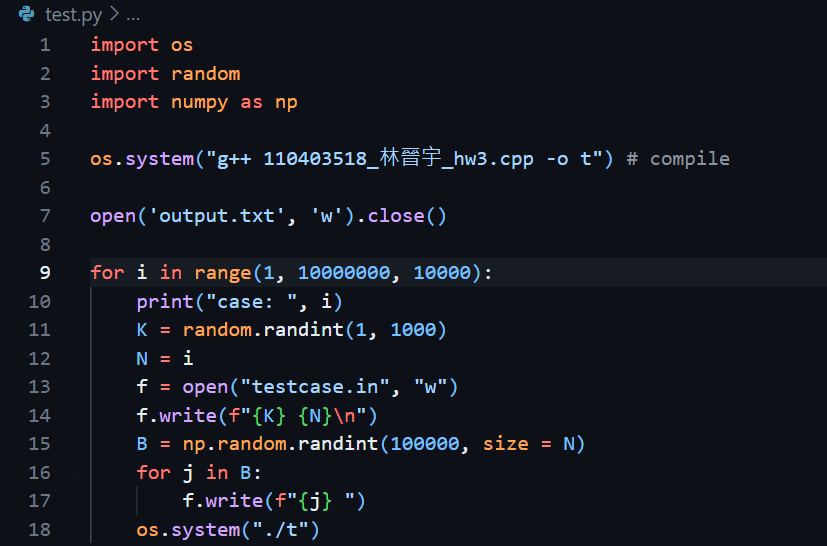
        }

    }

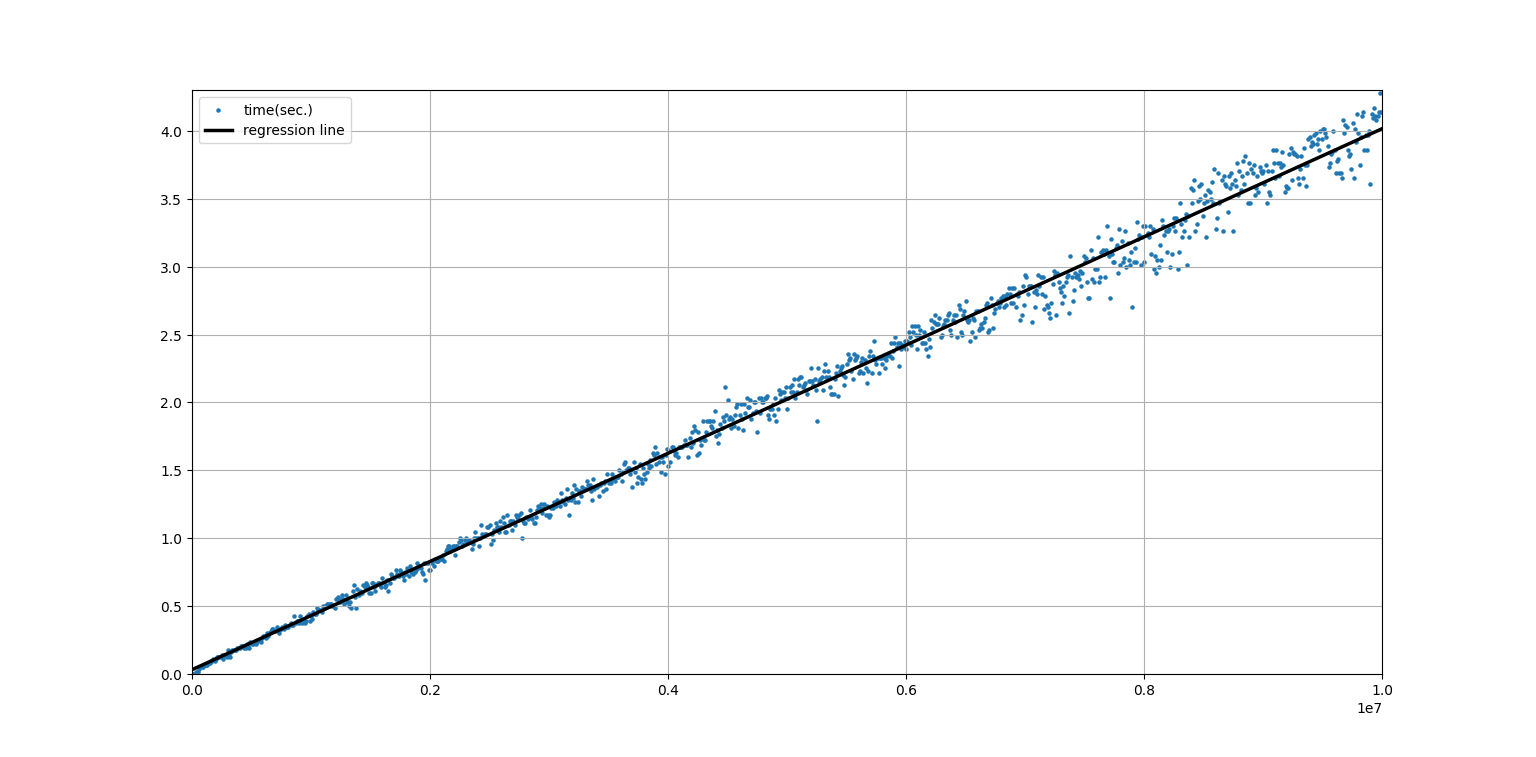
1. **分析複雜度**

* 時間複雜度: 一共有n個request，每次request需要將資訊push進priority\_queue，時間為**O(nlogn)**。
* 空間複雜度: **O(n)**。

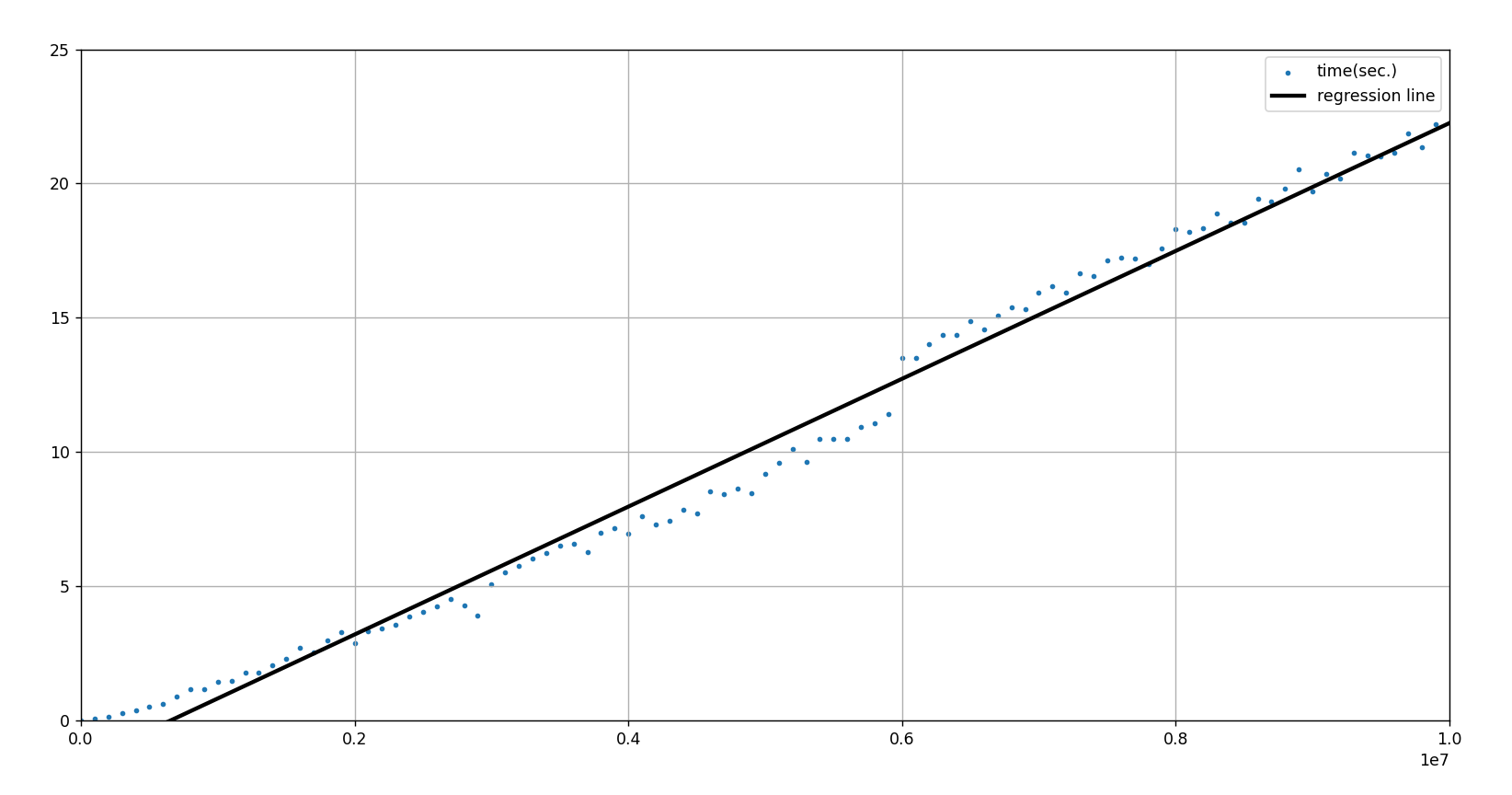
1. **如何生成測資**

****

以上為生成測資的python程式碼，**n**從1開始慢慢地增至10^7(每次增加10000)，**k**及**b**皆為隨機產生，總共產出1000筆資料。

1. **結果**

首先，我將b設為1-100000的隨機數，認為這樣比較容易發生hit，將1000筆測資讀入程式後，將每一次的執行時間畫成圖表，橫軸為N的大小，縱軸為時間(秒數)，藍點為實際時間，黑線為回歸線，可以看到當**N大至10^7時，執行時間大約需要四秒**。



再者，我將b設為1-2147483647，將100筆測資毒入程式，時間明顯比第一次慢很多，**可以看到當N來到10^7時，時間大概在22秒**，雖然時間複雜度看起來只跟N有關，但實際上b的範圍也對時間影響很大，我猜想可能是priority\_queue在每次push及排序時要花費更多時間。