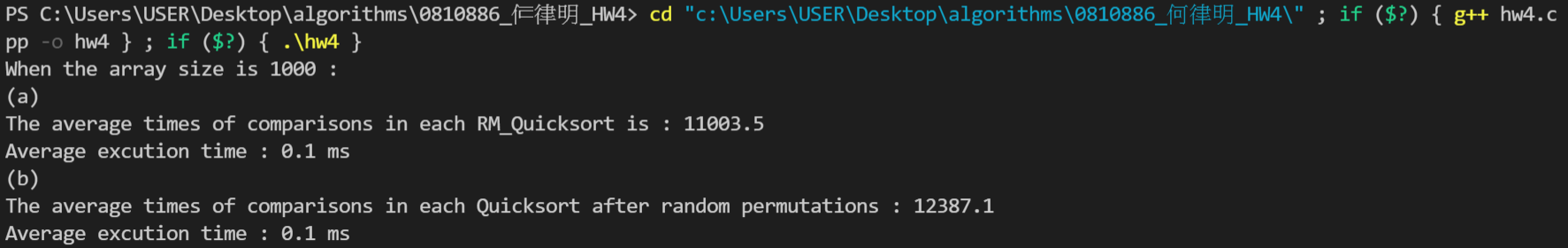
HW4

 (a)

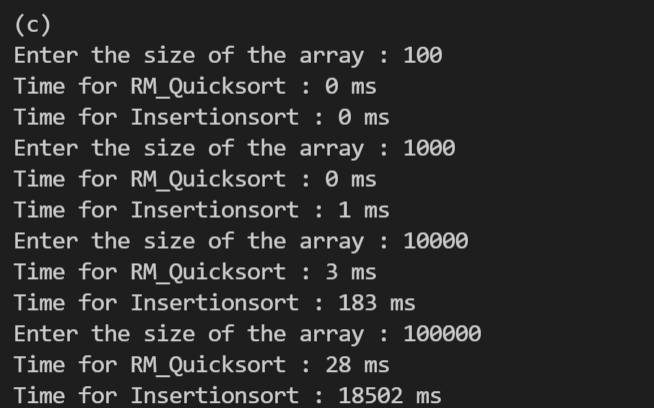
在一個有1000筆尚未被排序資料的陣列中，我總共做了10個1000筆資料的陣列，去取他們調用一次Randomized Quicksort函數在Partition函數中的第4行比較次數，結果平均大約為11000次，如上圖:

(b)

要求我們先把陣列隨機排序再做Quicksort，跟直接把陣列去做Randomized Quicksort，他們的結果不論是執行時間或是比較次數其實都差不多，我想原因為因為Quicksort函數執行有無效率判斷依據為pivot的位置，而選的pivot越中間，執行效率就越好，畫出來的tree也就越對稱，深度也因此降低，而要求我們先把陣列隨機排序再做Quicksort，跟直接把陣列去做Randomized Quicksort都為了不要拿到最不好的pivot而去做隨機挑選的動作，差別在於一個直接隨機挑選陣列其中一個element為pivot，另一個則是隨機把陣列打亂然後把最後一個當pivot，本質上都是隨機，也因此比較的次數、執行時間接近也實屬合理。

(c)

第三題的部分我是拿Insertion sort來跟這次的Quicksort做比較，如下圖:



很明顯的Insertion sort做排序所花的時間相比Quicksort(RM\_Quicksort)來說真的是非常差距之大，根據作業一的結論Insertion sort的執行時間Big O為n平方，相較之下，Quicksort(RM\_Quicksort)為n\*log(n)。

Code Introduction

void Quicksort(vector <int> &A, int p, int r){

    if (p < r){

        int q = Partition(A, p, r);

        Quicksort(A, p, q-1);

        Quicksort(A, q+1, r);

    }

}

Quicksort，照著課本pseudo code。

int Partition(vector <int> &A, int p, int r){

    int x = A[r];

    int i = p - 1; //Start from -1

    for (int j=p ; j<r ; ++j){

        timesofcomparisons++;

        if(A[j] <= x){

            ++i;

            //Swap

            int tmp = A[i];

            A[i] = A[j];

            A[j] = tmp;

        }

    }

    int temp = A[i+1];

    A[i+1] = A[r];

    A[r] = temp;

    return i+1;

}

Partition，根據課本pseudo code。

void RM\_Quicksort(vector <int> &A, int p, int r){

    if (p < r){

        int q = RM\_Partition(A, p, r);

        RM\_Quicksort(A, p, q-1);

        RM\_Quicksort(A, q+1, r);

    }

}

RM\_Quicksort，根據課本pseudo code。

int RM\_Partition(vector <int> &A, int p, int r){

    //Random(p, r)

    int sz = r-p+1;

    int i;

    if (sz >= 20000){

        int part = (sz/20000);

        if(part < sz/20000.0)

            part++;

        int randompart = rand()%(part);

        int remain = sz - (part-1)\*20000;

        int randremain = rand()%(remain);

        int randomnumber = randompart\*20000 + randremain;

        i = randomnumber + p;

    }

    else{

        i = rand()%(sz) + p;

    }

    i = rand()%(sz) + p;

    int tmp = A[i];

    A[i] = A[r];

    A[r] = tmp;

    return Partition(A, p, r);

}

RM\_Partition，根據課本pseudo code。

void random\_in\_size(vector<int> &v){

    int sz = v.size();

    int part = (sz/20000);

    if(part < sz/20000.0)

        part++;

    for(int i=0 ; i<v.size() ; ++i){

        int randompart = rand()%(part);

        int remain = sz - (part-1)\*20000;

        int randremain = rand()%(remain);

        int randomnumber = randompart\*20000 + randremain;

        //Swap v[i] with v[randomnumber]

        int temp  = v[i];

        v[i] = v[randomnumber];

        v[randomnumber] = temp;

    }

}

Random in size，前次作業出現過的function，能把陣列中的element隨機打亂。

void Insertionsort(vector <int> &arr){

    int s = arr.size();

    for(int i=1 ; i<s ; ++i){

        int key = arr[i];

        int j = i-1;

        while(j>=0 && key<arr[j]){

            arr[j+1] = arr[j];

            j--;

        }

        arr[j+1] = key;

    }

}

第一次作業有出現過的Insertion sort，一樣能把陣列排序，簡單但耗時。

void copyandnew(vector <int> &a, vector <int> &b){

    int sz = b.size();

    for(int i=0 ; i<sz ; ++i){

        a.push\_back(b[i]);

    }

}

複製一個一模一樣的vector陣列(from b to a)。