

附圖為input size為100 &1000 & 10000 & 100000的insertion sort & merge sort所花的時間。

Running time comparison:

Input size 100:

Insertion sort : 0 s

Merger sort : 0 s

Input size 1000:

Insertion sort : 0 s

Merger sort : 0 s

Input size 10000:

Insertion sort : 0.07 s

Merger sort : 0.001 s

Input size 100000:

Insertion sort : 5.753 s

Merger sort : 0.017 s

User interface:

Input :

Enter the array size : ()🡨 user inputs the array size they want, precision to ms

Output:

----------------------------------

Time for insertion sort : () s 🡨 actual time for insertion sort, precision to ms

-----------------------------------

Time for merge sort : () S 🡨 actual time for merge sort, precision to ms

======================

Functions introduction:

//print the whole sorted array

void ptarr(int arr[], int s){

    for(int i=0 ; i<s ; ++i){

        cout<<arr[i]<<" ";

    }

    cout<<endl;

}

arr 為輸入的 array， s 為 array 的大小。

輸出整個array內容。

// Copy an array

void copyarr(int arr[], int tmp[], int s){

    for(int i=0 ; i<s ; ++i){

        tmp[i] = arr[i];

    }

}

arr 為要被複製的 array，s 為 array 的大小，tmp 為被複製完成的 array。

tmp跟arr為同內容的不同array。

//insertion sort

void insertionsort(int arr[], int s){

    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;

    }

}

arr 為輸入的 array， s 為 array 的大小。

arr則會被直接排序完。

// Merge-Sort recursively

void mergesort(int arr[], int p, int r){

    if(p < r){

        double q = floor((p+r)/2);

        mergesort(arr, p, q);

        mergesort(arr, q+1, r);

        merge(arr, p, q, r);

    }

}

arr 為輸入的 array，p為要被排序的起始index值，q為要被排序的最後index值。用recursive的方法進行排序。

// Merge

void merge(int arr[], int p, int q, int r){

    int n1=q-p+1, n2=r-q;

    int L[n1], R[n2];

    for (int i = 0; i < n1; ++i){

        L[i] = arr[p+i];

    }

    for (int i = 0; i < n2; ++i){

        R[i] = arr[q+1+i];

    }

    int i=0, j=0;

    for (int k=p; k <= r; ++k){

        // R is empty or L <= R, move L to arr

        if ((j==n2 || L[i]<=R[j]) && i!=n1){

            arr[k] = L[i];

            ++i;

        }

        // L is empty or L > R, move R to arr

        else if((i==n1 || L[i]>R[j]) && j!=n2){

            arr[k] = R[j];

            ++j;

        }

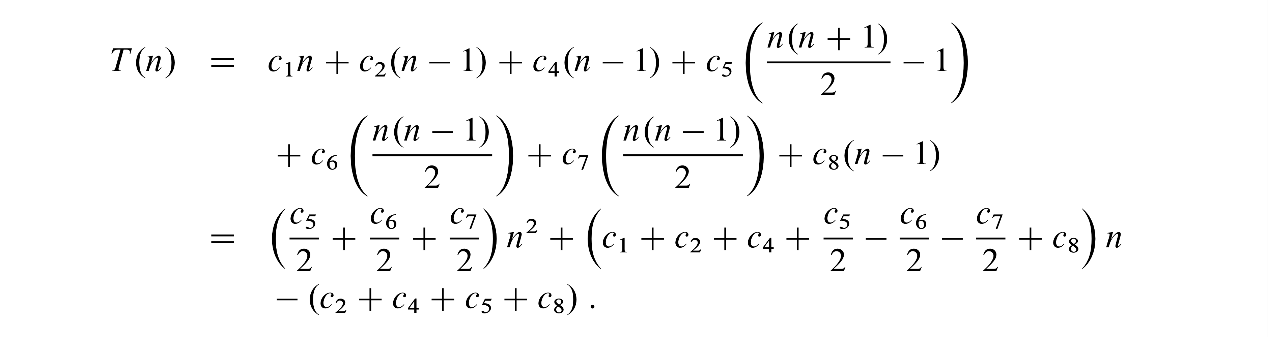
    }

}

arr 為輸入的 array，p為要被merge的第一個array起始index值，q為要被merge的第一個array的最後index值，r為要被merge的第二個array的最後index值。

兩個array會被排序好放在arr的相對應位置。

Observation:

我的程式碼都是根據課本的pseudo code去打的。所以running time 應該課本描述得差不多，以下為insertion sort’s worst case 的running time 

基本上worst case & average case running time 都是quadratic ，具有二次項，所以當input size 很大且為原本10倍時，running time幾乎是原本的n平方倍，也就是100倍(或許有係數，但我就先以比例來觀察 big O)，而insertion sort的best case則為一次項，因為我用的計時方法是library <ctime>裡的clock() 函數，精確性不太算高，只精確到毫秒等級。當n = 100跟n = 1000時，因為速度都太快，所以顯示出來的為0秒，我自己平常在試的時候，有時候n = 1000的running time為0.001秒。但n = 10000時，insertion sort的running time主要大概落在0.05 ~ 0.07秒之間，而n = 100000，大概落在5 ~ 7秒之間，差不多是nlogn ~ n平方之間的倍數關係，也符合理論值。

以下為我額外試的insertion sort，當n = 150000

Enter the array size : 150000

------------------------

Time for insertion sort :13.037 s

------------------------

========================

跟merge sort相比起來時間多了不少。

Merge sort是用recursive來實現，共有log(n)層，每層有n筆資料要處理，所以大約的running time大約是n\*log(n)，當n = 100時，理論上就跟n平方差了10倍，當然說running time絕對不是nlog(n)這麼簡單，可能還有一些常數去影響效率(真正的時間)。經過我的測試，當n從10000到100000，running time從0.001變到0.017差了17倍左右，也跟nlog(n)的比例差不多。

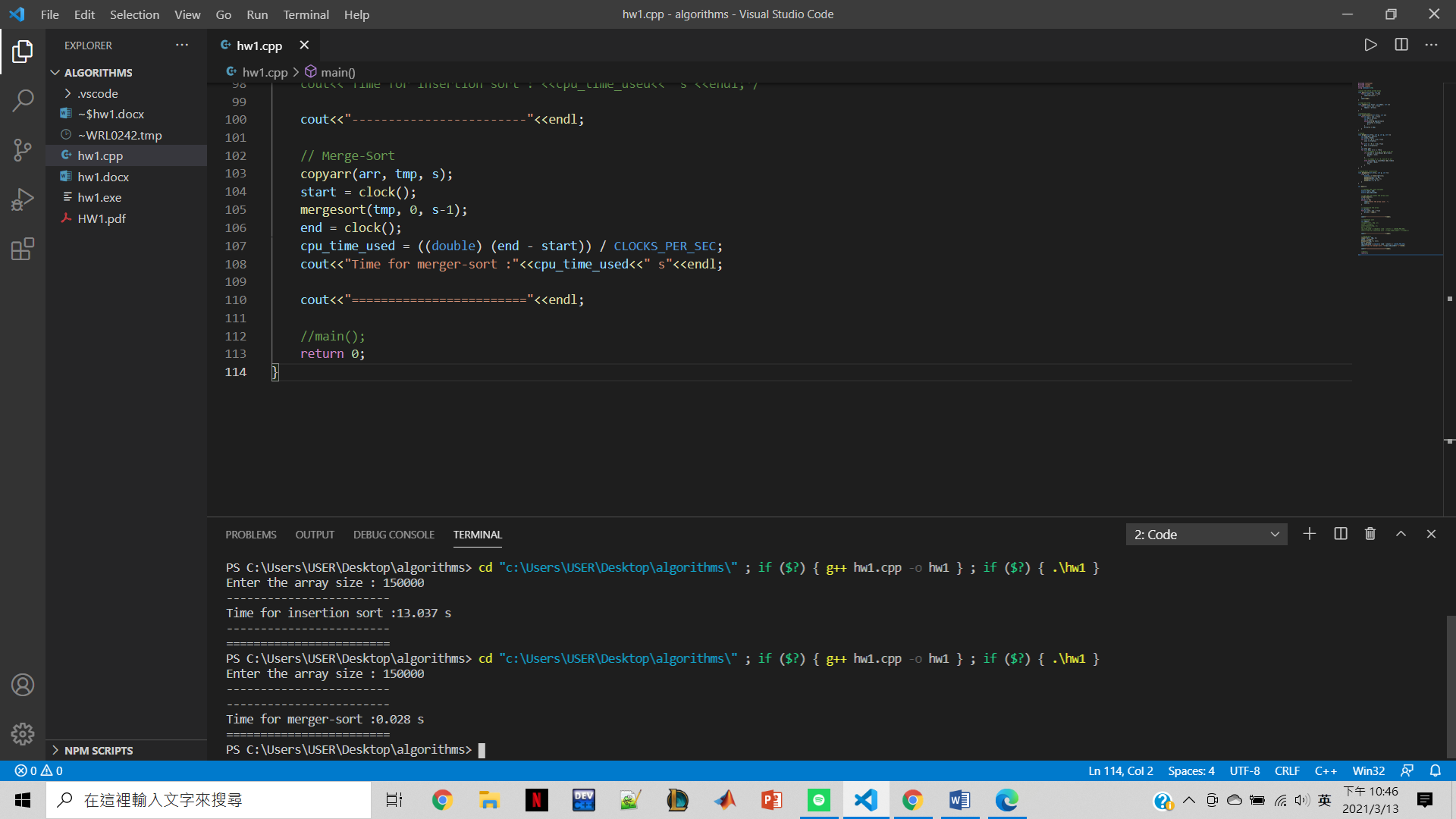
Enter the array size : 150000

------------------------

------------------------

Time for merge-sort :0.028 s

========================



0.028/0.017 = 1.64705…

150000\*log(150000) = 776413

100000\*log(100000) = 500000

776413/500000 = 1.552

🡪1.64705跟1.552在running time的角度來皆看非常符合n\*log(n)，也驗證了我們上課中說的理論。

雖然說recursive在寫程式來說不太有效率，因為他需要暫存空間，也需要一層一層來代回去上一層的時間，雖說為n\*log(n)，但其實還有更快的方法，像quick sort也是n\*log(n) //log(n)層，每層n筆資料，但卻不需要用到recursive。

Conclusion:

Insertion sort的big O為n square，merge sort的big O為n\*log(n)。