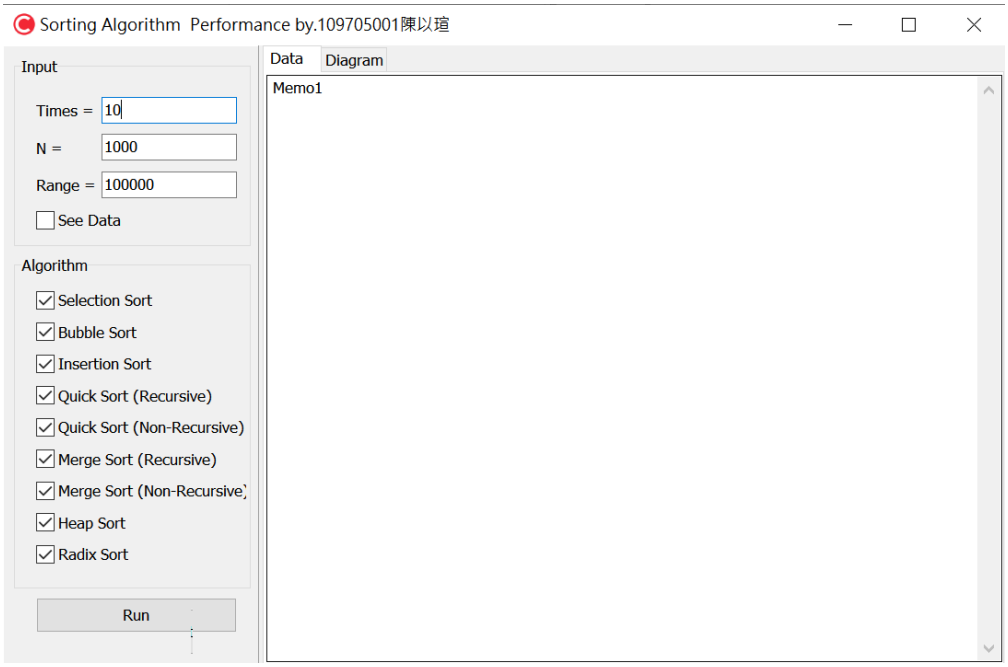
HW09_Sorting Algorithm Performance 作業報告

學號：109705001 姓名：陳以瑄

操作介面



完成的基本要求

1. **輸入整數 n 和希望產生的亂數範圍 range**
   程式碼: SortingAlgorithm.cpp 中第 32 行至第 42 行的部分
2. **輸入整數 Times 為希望執行的次數**
   程式碼: SortingAlgorithm.cpp 中第 310 行開始的部分(迴圈在第 325 行)
3. **實作下列排序演算法**

(1.) Selection sort

程式碼:SortingAlgorithm.cpp 中第 53 行至第 64 行的部分

```
//SelectionSort
void SelectionSort (int * data, int num)
{
    int min,tmp=0;
    for(j=0;j<num;j++)
    {
        min = j;
        for(k=j;k<num;k++)
            if(data[k]<data[min])
                min=k;
        if(min!=j)SWAP(data[min],data[j]);
    }
}
```

(2.) Bubble sort

程式碼:SortingAlgorithm.cpp 中第 67 行至第 77 行的部分

```
//BubbleSort
void BubbleSort (int * data, int num)
{
    int min,tmp=0;
    for(j=0;j<num;j++)
    {
        min = j;
        for(k=j;k<num;k++)
            if(data[k]<data[min])
                SWAP(data[min],data[k]);
    }
}
```

(3.) Insertion sort

程式碼:SortingAlgorithm.cpp 中
第 80 行至第 94 行的部分

```
//InsertionSort
void InsertionSort (int * data, int num)
{
    int x;
    for(j=1;j<num;j++)
    {
        x=data[j];
        k=j;
        while(k>0&&x<data[k-1])
        {
            data[k]=data[k-1];
            k--;
        }
        data[k]=x;
    }
}
```

(4.) Quick sort (recursive versions)

程式碼:SortingAlgorithm.cpp 中
第 97 行至第 100 行的部分

```
//QuickSort (Recursive)
void QuickSort (int * data, int left,int right)
{
    int tmpi,tmpj,target,tmp=0;
    if(left<right)
    {
        tmpi=left+1;
        tmpj=right;
        target=data[left];
        do
        {
            while(tmpi<=tmpj&&data[tmpi]<target)tmpi++;
            while(tmpi<=tmpj&&data[tmpj]>=target)tmpj--;
            if(tmpi<tmpj)SWAP(data[tmpi],data[tmpj]);
        }while(tmpi<tmpj);
        if(left<tmpj)SWAP(data[left],data[tmpj]);
        QuickSort(data,left,tmpj-1);
        QuickSort(data,tmpj+1,right);
    }
}
```

(5.) Quick sort (non-recursive versions)

程式碼:SortingAlgorithm.cpp 中第 118 行至第 162 行的部分
(118 至 138 是堆疊的定義跟 push 和 pop 139 至 162 才是演算法的主要部分)

```
//QuickSort (Non-Recursive)
struct StackNode
{
    int l;
    int r;
    struct StackNode * next;
};
struct StackNode * top;
void PushStack( int left, int right)
{
    struct StackNode * old_top= top;
    top =(struct StackNode*)malloc(sizeof(struct StackNode));
    top->l=left;
    top->r=right;
    top->next=old_top;
}
struct StackNode * PopStack()
{
    struct StackNode * old_top= top;
    top=old_top->next;
    return old_top;
}
```

```
void QuickSortNon (int * data, int left,int right)
{
    top=NULL;
    PushStack(left,right);
    while(top!=NULL)
    {
        struct StackNode * node= PopStack();
        left=node->l;
        right=node->r;
        int target = data[left];
        int tmpi=left+1;
        int tmpj = right;
        int tmp;
        do
        {
            while(tmpi<=tmpj&&data[tmpi]<target)tmpi++;
            while(tmpi<=tmpj&&data[tmpj]>=target)tmpj--;
            if(tmpi<tmpj)SWAP(data[tmpi],data[tmpj]);
        }while(tmpi<tmpj);
        if(left<tmpj)SWAP(data[left],data[tmpj]);
        if(left<tmpj-1)PushStack(left,tmpj-1);
        if(tmpj+1<right)PushStack(tmpj+1,right);
    }
}
```

(6.) Merge sort (recursive versions)

程式碼:SortingAlgorithm.cpp 中第 165 行至第 196 行的部分

(165 至 185 是 Merge 兩個以排序的陣列 139 至 196 是 Merge Sort)

```
//MergeSort (Recursive)
void Merge(int *C,int c,int *A,int a,int m,int*B,int b,int o)
{
    int *tmpArray=(int*)malloc((o+1)*sizeof(int));
    int p;
    for(p=a;p<=m;p++)
        tmpArray[p]=A[p];
    for(p=b;p<=o;p++)
        tmpArray[p]=B[p];
    while(a<=m&&b<=o)
    {
        if(tmpArray[a]<tmpArray[b])
            C[c++]=tmpArray[a++];
        else
            C[c++]=tmpArray[b++];
    }
    while(a<=m)
        C[c++]=tmpArray[a++];
    while(b<=o)
        C[c++]=tmpArray[b++];

}
```

```
void MergeSort(int * data, int left,int right)
{
    int m;
    if(left<right)
    {
        m=(left+right)/2;
        MergeSort(data,left,m);
        MergeSort(data,m+1,right);
        Merge(data,left,data,left,m,data,m+1,right);
    }
}
```

(7.) Merge sort (non-recursive versions)

程式碼:SortingAlgorithm.cpp 中第 198 行至第 214 行的部分

```
//MergeSort (NonRecursive)
void MergeSortNon(int * data, int num)
{
    int len=1,tmpi;
    while(len<=num)
    {
        tmpi=0;
        while(tmpi<num-len)
        {
            if(tmpi+2*len<num)
                Merge(data,tmpi,data,tmpi,tmpi+len-1,data,tmpi+len,tmpi+len*2-1);
            else
                Merge(data,tmpi,data,tmpi,tmpi+len-1,data,tmpi+len,num-1);
            tmpi+=2*len;
        }
        len*=2;
    }
}
```

(8.) Heap sort

程式碼:SortingAlgorithm.cpp 中第 217 行至第 250 行的部分

(217 至 1232 是堆積的 Restore 233 至 250 是 Heap Sort)

```
//HeapSort
int* Restore(int * data,int s, int r)
{
    int tmpi=s,tmpj,tmp;
    while(tmpi<=r/2)
    {
        if(tmpi*2+1>r||data[tmpi*2]<=data[tmpi*2+1])
            tmpj=tmpi*2;
        else
            tmpj=tmpi*2+1;
        if(data[tmpi]<=data[tmpj])
            break;
        SWAP(data[tmpj],data[tmpi]);
        tmpi=tmpj;
    }
    return data;
}
```

```
void HeapSort(int * data,int num)
{
    int *tmpArray=(int*)malloc((num+1)*sizeof(int));
    int j;
    for(j=0;j<num;j++)
        tmpArray[j+1]=data[j];
    for(j=num/2;j>=1;j--)
    {
        tmpArray=Restore(tmpArray,j,num);
    }
    for(j=num;j>1;j--)
    {
        SortData[num-j]=tmpArray[1];
        tmpArray[1]=tmpArray[j];
        tmpArray=Restore(tmpArray,1,j-1);
    }
    SortData[num-1]=tmpArray[1];
}
```

(9.) Heap sort

程式碼:SortingAlgorithm.cpp 中第 253 行至第 300 行的部分

```
//RadixSort
void RadixSort(int * data,int num)
{
    int max = 0,tmpi,radix=0,digit;
    int *tmpArray=(int*)malloc((num)*sizeof(int));
    int *count=(int*)malloc(10*sizeof(int));
    int *temp=(int*)malloc((num)*sizeof(int));
    int *index=(int*)malloc(10*sizeof(int));
    for (tmpi=0; tmpi<num; tmpi++)
    {
        tmpArray[tmpi]=data[tmpi];
        if (data[tmpi]>max)
            max = data[tmpi];
    }
    while(max!=0)
    {
        radix++;
        max/=10;
    }
    for (tmpi=1; tmpi<=radix; tmpi++)
    {
        for (j=0; j<10; j++)
            count[j]=0;
        for (j=0; j<num; j++)
        {
            digit = tmpArray[j]%10;
            count[digit]++;
        }
        index[0]=0;
        for (j=1;j<10;j++)
            index[j]=index[j-1]+count[j-1];
        for (j=0; j<num; j++)
        {
            digit = tmpArray[j]%10;
            temp[index[digit]++] = data[j];
        }
        for (j=0; j<num; j++)
        {
            tmpArray[j]=data[j] = temp[j];
        }
        int tmpk=tmpi;
        while(tmpk>0)
        {
            for (j=0; j<num; j++)
                tmpArray[j]/=10;
            tmpk--;
        }
    }
}
```

4. 印出 Sort 後的 n 個亂數並自動檢測是否成功排序與執行各排序演算法的 CPU 時間

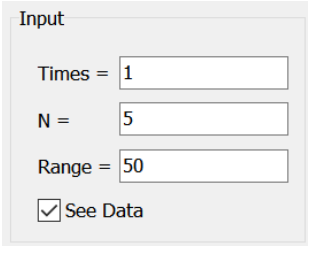
->如果勾選 see data 會顯示生成出的 data 跟排序後的 data

自行檢測程式碼(在第 302 行至 309)

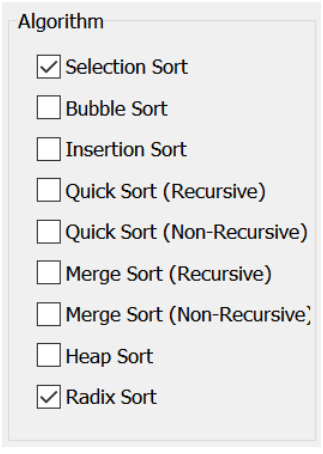
```
int SelfCheck(int* data)
{
    int p;
    for(p=1;p<num;p++)
        if(data[p]<data[p-1])
            return 0;
    return 1;
}
```

執行結果(有勾選 See data)

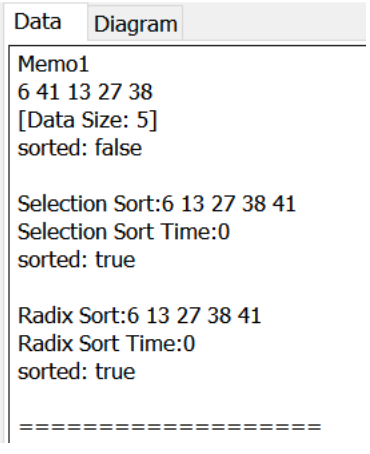
Data 設定



演算法設定



按下 Run Button 後在右側 Data 頁面

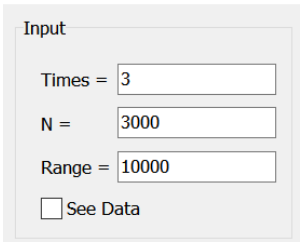


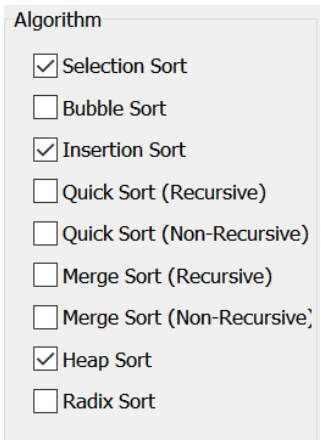
5. 比較其資料數量不同時，上述演算法的 CPU 執行時間

->按下 Run 後詳細時間呈現在 Data 頁面

執行結果(有勾選 See data)

Data 設定　　**演算法設定**　　**按下 Run Button 後在右側 Data 頁面**






```
[Data Size: 3000]
sorted: false

Selection Sort Time:0.0160000007599592
sorted: true

Insertion Sort Time:0
sorted: true

Heap Sort Time:0
sorted: true

====================
[Data Size: 6000]
sorted: false

Selection Sort Time:0.0469999983906746
sorted: true

Insertion Sort Time:0.0320000015199184
sorted: true

Heap Sort Time:0
sorted: true

====================
[Data Size: 9000]
sorted: false

Selection Sort Time:0.0939999967813492
sorted: true

Insertion Sort Time:0.0469999983906746
sorted: true

Heap Sort Time:0
sorted: true

====================
```

6. 利用 BCB TChart 元件畫出各排序演算法的執行效能折線圖

->按下 Run 後在 Diagram 頁面會呈現比較圖(有參與的演算法才會呈現)

執行結果