演算法分析期末報告

排序視覺化展示 - Sorting Visualization

學生：高沛功 101971018
指導教授：何瑁鎧 教授

2013年秋季班

排序視覺化展示 - Sorting Visualization

學生：高沛功 101971018
指導教授：何瑁鎧 教授

Abstract

排序演算法已被研究多年並已成熟。在學習過程中只須吸收成果即可，這很方便可是同時也是問題所在。就因為排序演算法已太成熟所以反而沒人在乎它在底是如何運作的，一切都認為理所當然。此份報告的目的是將這些已成熟的排序演算法以視覺化呈現並展示其運作過程，以協助學生們理解該排序演算法的特性，理解這些排序演算法的神奇之處。同時向過去發展這些演算法的先驅們致敬。

Introduction

排序在計算機科學的歷史中，已被大量研究也被認為是已解決的問題。雖然大部分人認為這是一個已經被解決的問題，有用的新演算法仍在不斷的被發明。（例：圖書館排序在2004年被發表）

會特別做這項目是為了想要了解排序的數據搬運過程，可以用視覺觀察而不只是靠想像力。重點將在排序過程可以看得見，速度度量以排序動作的次數來評估，速度最好的排序法其排序動作也應該更少。排序動作有比較、交換、移動等。

排序法發展到現在已有非常多的選擇。在時間有限狀況下挑選了10個排序法。選擇的依據像是直覺想到的方法，如：泡沫排序法、插入排序法、選擇排序法。當然公被認多年速度最快的快速排序法也要入列。有外部排序特性的合併排序法也是觀察目標之一。也選擇了在理論上最快的線性排序法中的計數排序法、基數排序法來觀察

之後也會比較這些排序法的優劣，檢驗排序的結果與理論上的時間複雜度有否吻合。

2.Design and Implement

在開始實作之前，理所當然的先限定一些需求範圍。如下：

一、排序過程可以看到數據移動過程。

在實務上排序一般是在後端執行使用者在畫面上都是直接看到排序好的結果。

二、排序法選擇俱不同特性且經典的。

- 大腦直覺性聯想到的方法：插入排序法(Insertion Sort)、泡沫排序法(Bubble Sort)、選擇排序法(Selection Sort)。
- 已被公認是最快的排序法：快速排序法(Quick Sort)，與其改良版亂數快速排序法(Randomized Quick Sort)。
- 有研究過才出現的排序法：堆積排序法(Heap Sort)。
- 有名又經典的外部排序法：合併排序法(Merge Sort)、與其另一實作版本下上合併排序法(Bottom Up Merge Sort)。
- 理論上最快的線性排序法：計數排序法(Counting Sort)、基數排序法(Radix Sort)。

三、不同的排序法其在畫面呈現的方式要一致化。

這樣才有同一基礎比較的視覺感覺。

四、排序過程中，數據移動的延遲時間可以設定。

用以感受不同排序法的好壞速度感。好與不好的排序法在不同延遲時間可以明顯感受到，因其數據搬動、比較的次數會差很多。

五、可以統計排序過程中做了多少次“排序動作(sorting action)”。

排序動作是指完成排序工作要作的動作，這些動作大概可以分成四類，比較、交換、移動與定址計算，說明如下：

- 比較動作

  數列中數據比較後才能依大小重新排列。一般只是讀取未寫入所以速度應該最快的，而比較是排序很動要的依據當然要列入排序動作計算次數。在次數上只計1次。

- 交換動作

  兩筆數據交換位置，也是in place 排序法數據位置變更的方法。在程式上一般要3次assignment，但在次數上只計1次。

- 移動動作

  一筆數據搬移到另一地方。外部排序法的數據位置變更的方法，如：合併排序法。在次數計算上也只計1次。

- 定址計算動作

  計數排序法與基數排序法的排序方式很特別，先把數據的位置計算好，再把

數據搬過去。與一般比較後調整位置方法完全不同。在時間上應該也是很短的，在此因認為定址計算是排序過程中重要的步驟故列入計算。在次數上計1次。

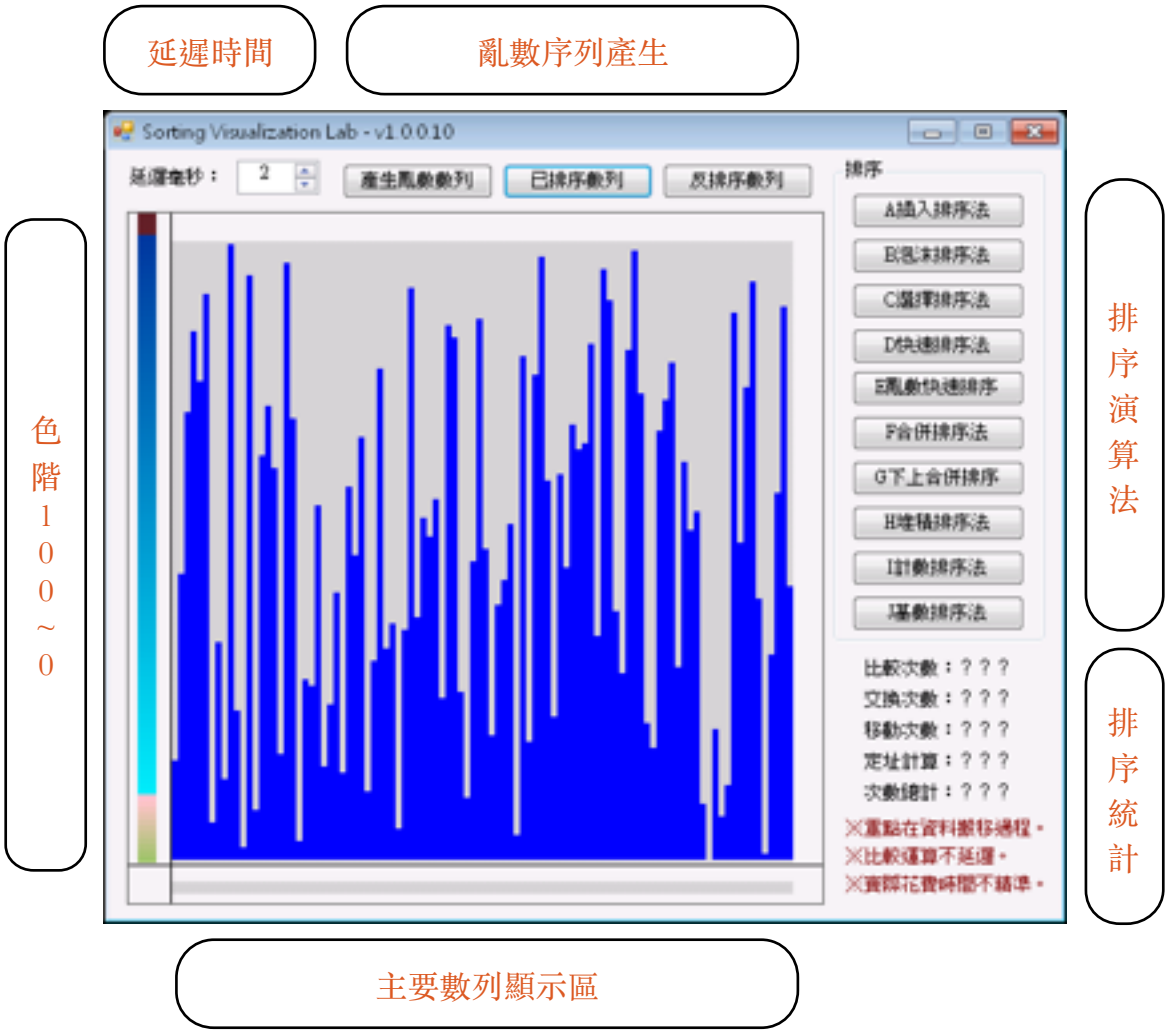
此四項排序動作外的部份，大概都是鷹架碼或一些局部的雜項，對於排序效率影響不大不列入計算。

六、數列排序前的狀態可以提供三種類型：亂序數列、已排序數列與反向已排序數列。目的為用來測試已排好與未排好的狀態下各排序演算法的表現是更好或更壞。

七、畫面呈現上，對於搬移次數多與少可以以產色深淺或色階方式呈現，這樣可以一目了然大致做了多或少的排序動作。

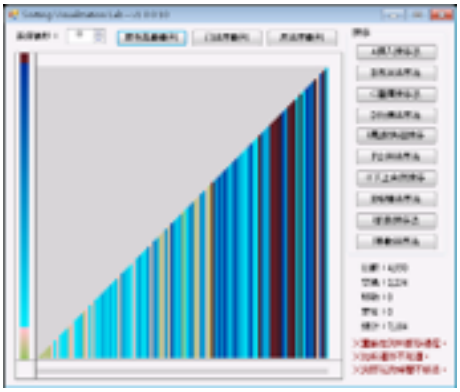
3.Harvest

下圖為完成畫面與其佈局說明。

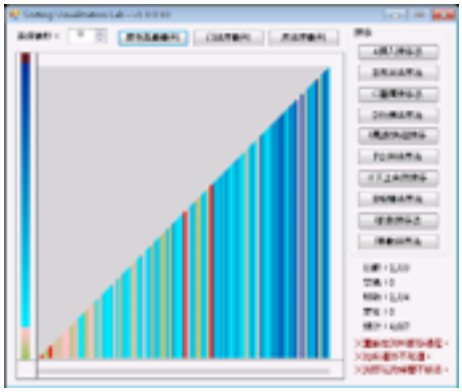


下面附上一些執行畫面。

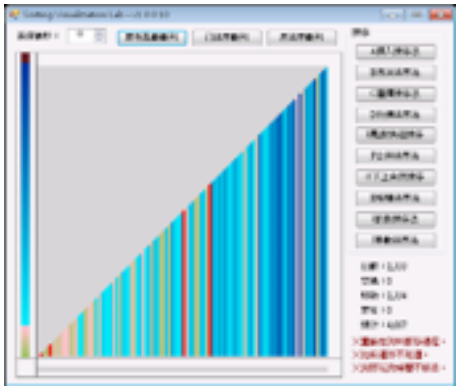
Bubble sort - 亂序起始



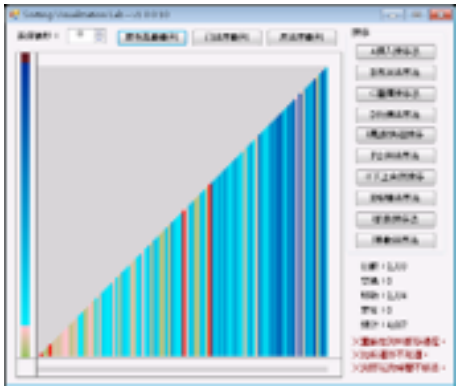
Insertion sort - 亂序起始



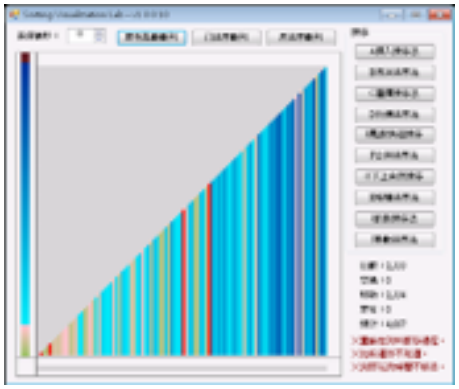
Quicksort - 亂序起始



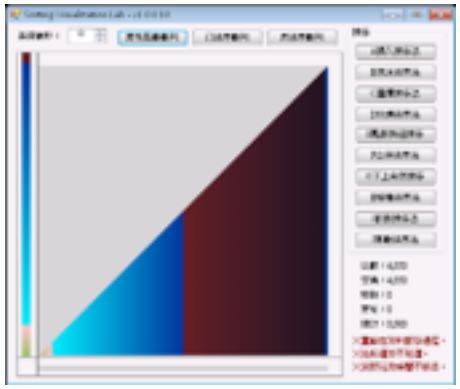
Randomized Quicksort - 亂序起始



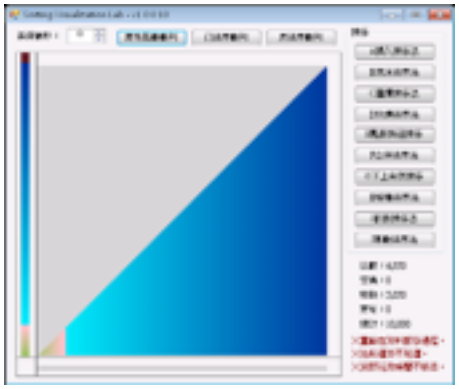
Selection sort - 亂序起始



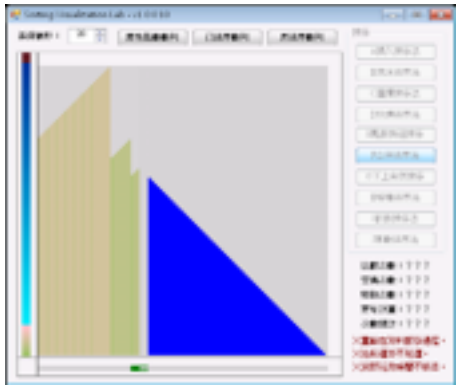
Bubble sort - 反排序起始



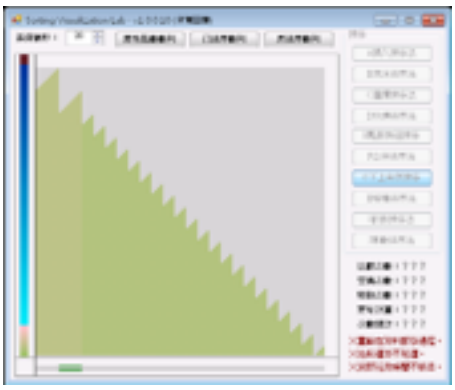
Insertion sort - 反排序起始



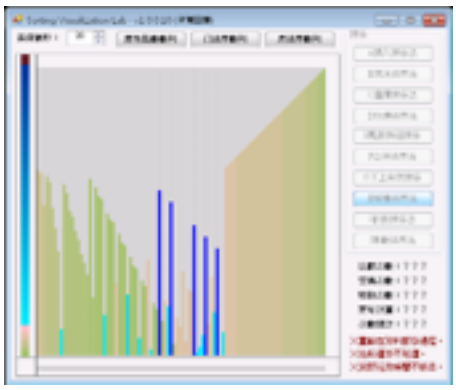
Merge sort - 反排序起始



Bottom Up Merge sort - 反排序起始



Heap sort - 反排序起始



4.Sorting Algorithm Compare

在比較時首先想知道實際與理論上速度表現是否一致。這次只比較的時間部份。現在的電腦記憶體都很大，空間複雜度在大部份狀況下是不須考量的。

下表為這次選用的排序演算法的時間複雜度。

Name	Best	Average	Worst	Remark
Insertion sort	n	n^2	n^2	
Bubble sort	n^2	n^2	n^2	
Selection sort	n^2	n^2	n^2	
Quicksort	n log n	n log n	n^2	
Randomized Quicksort	n log n	n log n	n^2 (less likely)	expected time = n log n
Merge sort	n log n	n log n	n log n	
Bottom Up Merge sort	n log n	n log n	n log n	
Heapsort	n log n	n log n	n log n	
Counting sort	n + r	n + r	n + r	r is the range of numbers to be sorted
Radix sort	n * m	n * m	n * m	with bucket sort m = log n

這些理論上的時間複雜度與這次做出來的結果，在多次試跑下完全與實際況狀吻合。

其中最有意思的是Quicksort 與 Randomized Quicksort 的比較。在一般的亂序數列下，Randomized Quicksort的執行時間比Quicksort 稍慢一些些。但在已排序數列下，Randomized Quicksort的執行時間也可以是 n log n 遠遠把Quicksort 拋在腦後。雖然在理論上Randomized Quicksort 與Quicksort 在worst 都是 n^2 但實際上因亂數選取 partition 所以除非真的非常非常不幸，不然都會在 n log n 內完成。

而 Merge sort 與 Heapsort的表現跟預期一樣的穩定，不管在亂序或已排序都一樣。在理論上雖然都是 n log n 但在實際上 Mergesort 的表現比Heapsort 好上很多。

Merge sort 在版本上又分Top Down 與 Bottom Up兩種。試跑多次下兩方的表現不分軒輊。在選擇上個人會偏愛 Bottom Up 版本，原因有二：一、非遞迴；二、同一個合併層級可以同時執行非常適合平行處理。

而那三個基本的排序法，Insertion Sort、Bubble sort 與 Selection sort 效能果然很糟糕。其中因為比較動作不加入延遲時間所以可能誤認為 Selection sort 的效能不錯，

但實際上其比較動作的次數是相當高的。Bubble sort 是所有排序法表現最差的。而 Insertion sort 若是用在 linked list 效果應該不差，只要數列不長的話。

而這次特別把線性排序法加入這次專案，因為個人實在很難想像為何理論上最快的線性排序法O(n)在實際上會輸給O(n log n)的Quicksort。

在多次試跑後個人認為Counting sort 與 Radix sort 比 Quicksort 更快。我想主要是因為其數據的排序位址是用算的而不是比較再調整出來的。但因其特性在實務應用上卻很難派上場。

首先查看Counting sort，此法在實務上用處極少，因要把值(value)變成索引(index)對很多數據來說是不太可能的。Radix sort 則要看其所搭配的stable sorts，在此是選用bucket sort。在實務上Radix sort 也有與Counting sort 同樣的難題，也有人提出一些解法，但個人認為那只些都是特解非一般解。在實務上這兩個方法只能用在特定的數據種類，比如整數。

6.Conclusion

結論就是：老師是對的。先驅們在排序方面的研究再次被驗證了一次。

這次排序的驗證結果，實測結果與理論是完全吻合的。也是大家一直都認為認定的結果。即然如此為何還要去測試呢。雖所有資料都指向“就是這樣”，可是個人還是小有疑惑。

在完成這份報告時，剛好也發現有人也在驗證排序演算法：

> Timo Bingmann - 15 Sorting Algorithms in 6 Minutes
> http://www.youtube.com/watch?v=kPRA0W1kECg

最後，我想引用先哲的一句話：「實踐是檢驗真理的唯一標準」。

References

附上這些排序法的程式碼。

wikipedia - Sorting algorithm
http://en.wikipedia.org/wiki/Sorting_algorithm
wikipedia - Quicksort
http://en.wikipedia.org/wiki/Quick_sort
wikipedia - Radix sort
http://en.wikipedia.org/wiki/Radix_sort
youtube - Timo Bingmann - 15 Sorting Algorithms in 6 Minutes
http://www.youtube.com/watch?v=kPRA0W1kECg
上課講義

原始碼下載網址 GitHub
https://github.com/relyky/winFrm_SortLab

Appendix

附上本專案排序程式碼

```
/// 泡沫排序法
public static void BubbleSort(AValue[] datas)
{
    for (int i = 0; i < datas.Length - 1; i++)
        for (int j = i + 1; j < datas.Length; j++)
            if (datas[i] > datas[j])
                Exchange(datas, i, j); // swap datas[i] <-> datas[j]
}

/// swap
public static void Exchange(AValue[] datas, int i, int j)
{
    // exchange
    AValue tmp = datas[i];
    datas[i] = datas[j];
    datas[i].IncreaseMoveTimes(); // count sorting action
    datas[j] = tmp;
    datas[j].IncreaseMoveTimes(); // count sorting action

    // for visualization exchange
    DrawData(datas, i);
    DrawData(datas, j);

    // counting
    _exchangeCount++;

    // wait a short time
    Thread.Sleep(Form1._sleepTimespan);
}
```

```
/// 插入排序法
public static void InsertionSort(AValue[] datas)
{
   AValue[] sortedDatas = new AValue[datas.Length];
   for (int i = 0; i < datas.Length; i++)
   {
      // 從未排序數列取出一元素
      AValue c = datas[i];

      // 由後往前和已排序數列元素比較，直到遇到不大於自己的元素並插入此元素之後；
      int insertIndex = i;
      for(; insertIndex > 0 && sortedDatas[insertIndex - 1] > c; insertIndex--);

      // 插入-shiftback
      for (int j = i; j > insertIndex; j--)
      {
         // sortedDatas[j] <- sortedDatas[j-1]
         Update(sortedDatas, j, sortedDatas[j-1]);
      }

      // 插入-new
      Update(sortedDatas, insertIndex, c, Pens.Red);
   }

   // copy back
   CopyArray(sortedDatas, 0, sortedDatas.Length - 1, datas);
}

/// 變更數值 datas[index] <- value
public static void Update(AValue[] datas, int index, AValue value)
{
   // update value
   datas[index] = value;
   datas[index].IncreaseMoveTimes();

   // visual update
   DrawData(datas, index);

   // counting
   _updateCount++;

   // wait a short time
   Thread.Sleep(Form1._sleepTimespan);
}
```

```
/// <summary>
/// 選擇排序法
/// </summary>
public static void SelectionSort(AValue[] datas)
{
   //# Index i, j, min
   int minIndex;
   for (int i = 0; i < datas.Length; i++)
   {
      minIndex = i;
      for (int j = i + 1; j < datas.Length; j++)
         if (datas[j] < datas[minIndex]) // compare
            minIndex = j;

      //# Exchange data[i] and data[min]
      if(i != minIndex)
         Exchange(datas, i, minIndex);
   }
}
```

```
/// <summary>
/// 快速排序法
/// </summary>
public static void QuickSort(AValue[] datas, int p, int r)
{
   if (p < r)
   {
      int q = QuickSortPartition(datas, p, r);
      QuickSort(datas, p, q - 1);
      QuickSort(datas, q + 1, r);
   }
}

public static int QuickSortPartition(AValue[] datas, int p, int r)
{
   AValue x = datas[r];
   int i = p - 1;
   for (int j = p; j < r; j++)
      if (datas[j] < x)
      {
         // i = i + 1;
         i++;
         // exchange A[i]<->A[j]
         Exchange(datas, i, j);
      }

   // exchange A[i+1] <-> A[r]
   i++;
   if(i != r)
      Exchange(datas, i, r);

   return i;
}
```

```
/// 亂數快速排序法
public static void QuickSortRand(AValue[] datas, int p, int r)
{
   if (p < r)
   {
      int q = QuickSortRandPartition(datas, p, r);
      QuickSortRand(datas, p, q - 1);
      QuickSortRand(datas, q + 1, r);
   }
}

public static int QuickSortRandPartition(AValue[] datas, int p, int r)
{
  // randomize
   Random rand = new Random();
   int i = p + rand.Next(r - p);
   Exchange(datas, r, i); // exchange A[r]<->A[i]
   return QuickSortPartition(datas, p, r);
}
```

```
/// <summary>
/// 合併排序法
/// </summary>
public static void TopDownMergeSort(AValue[] datas)
{
   AValue[] mergedDatas = new AValue[datas.Length];
   TopDownSplitMerge(datas, 0, datas.Length - 1, mergedDatas);
}

public static void TopDownSplitMerge(AValue[] datas, int iBegin, int iEnd,
        AValue[] mergedDatas)
{
   if (iEnd <= iBegin) // consider it sorted.
      return;

   int iMiddle = (iEnd + iBegin) / 2;
   TopDownSplitMerge(datas, iBegin, iMiddle, mergedDatas); // merge left
   TopDownSplitMerge(datas, iMiddle + 1, iEnd, mergedDatas); // merge right
   Merge(datas, iBegin, iMiddle, iEnd, mergedDatas);
   CopyArray(mergedDatas, iBegin, iEnd, datas); // copy back
}
```

```
public static void Merge(AValue[] A, int iBegin, int iMiddle, int iEnd, AValue[] B)
{
   //# before - for visualization
   ResetDataCanvas(iBegin, 0, iEnd, 99); // reset data canvas - for visualization
   DrawC(iBegin, iEnd, Pens.Green, true); // draw C canvas - for visualization

   //# do merge
   int i0 = iBegin;
   int i1 = iMiddle + 1;
   int j = iBegin;
   while (i0 <= iMiddle && i1 <= iEnd)
   {
      if (A[i0] <= A[i1])
      {
         DrawC(i0, Pens.LightGray); // reset C - for visualization
         // B[j++] = A[i0++];
         Update(B, j++, A[i0++]);
      }
      else
      {
         DrawC(i1, Pens.LightGray); // reset C - for visualization
         // B[j++] = A[i1++];
         Update(B, j++, A[i1++]);
      }
   }

   while (i0 <= iMiddle)
   {
      DrawC(i0, Pens.LightGray); // reset C - for visualization.
      //B[j++] = A[i0++];
      Update(B, j++, A[i0++]);
   }

   while (i1 <= iEnd)
   {
      DrawC(i1, Pens.LightGray); // reset C - for visualization.
      //B[j++] = A[i1++];
      Update(B, j++, A[i1++]);
   }
}
```

```
/// <summary>
/// 下上合併排序法
/// </summary>
/// <param name="datas"></param>
public static void BottomUpMergeSort(AValue[] datas)
{
   /* let 'datas' as array A */
   /* let 'mergedDatas' as array B*/
   AValue[] mergedDatas = new AValue[datas.Length];

   /* Each 1-element run in A is already "sorted". */
   /* Make successively longer sorted runs of length 2, 4, 8, 16... until whole
array is sorted. */
   int n = datas.Length - 1;
   for (int width = 1; width <= n; width = 2 * width)
   {
      /* Array A is full of runs of length width. */
      for (int i = 0; i <= n; i = i + 2 * width)
      {
         /* Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[] */
         /* or copy A[i:n-1] to B[] ( if(i+width >= n) ) */
         Merge(datas, i, Min(i + width - 1, n), Min(i + 2 * width - 1, n),
               mergedDatas);
      }

      /* Now work array B is full of runs of length 2*width. */
      /* Copy array B to array A for next iteration. */
      /* A more efficient implementation would swap the roles of A and B */
      CopyArray(mergedDatas, 0, n, datas);

      /* Now array A is full of runs of length 2*width. */
   }
}
```

```
/// 堆積排序法
public static void HeapSort(AValue[] datas)
{
   BuildMaxHeap(datas);
   int heap_size = datas.Length;
   for (int i = datas.Length - 1; i >= 1; i--)
   {
      Exchange(datas, i, 0); // 0 base array
      heap_size--;
      MaxHeapify(datas, heap_size, 0);
   }
}

public static void BuildMaxHeap(AValue[] datas)
{
   int heap_size = datas.Length;
   for (int i = (datas.Length - 1) / 2; i >= 0; i--)  //--- 0 base array
      MaxHeapify(datas, heap_size, i);
}

public static void MaxHeapify(AValue[] datas, int heap_size, int index)
{
   int l = HeapLeft(index);
   int r = HeapRight(index);

   int largest = index;
   if (l < heap_size && datas[l] > datas[largest])
      largest = l;
   if (r < heap_size && datas[r] > datas[largest])
      largest = r;

   if (largest != index)
   {
      Exchange(datas, index, largest); // exchange A[i] <-> A[largest]
      MaxHeapify(datas, heap_size, largest);
   }
}

public static int HeapLeft(int index)
{
  return ((index + 1) * 2 - 1); // --- 0 base
}

public static int HeapRight(int index)
{
  return HeapLeft(index) + 1;
}
```

```
/// <summary>
/// 計數排序法
/// </summary>
public static void CountingSort(AValue[] datas)
{
   //# duplation as A in place of datas[]. and let datas[] as B;
   AValue[] A = new AValue[datas.Length];
   CopyArray(datas, 0, datas.Length -1, A);

  //# step 1,2.
   int[] C = new int[A.Length];
   for (int i = 0; i < C.Length; i++) // k == length[A], in this case.
      C[i] = 0; // do C[i] <- 0

   AValueEx.ResetCountCanvas();  // show C - for visualizaton

   //# step 3,4.
   for (int i = 0; i < A.Length; i++)
   {
      // C[A[i]]++;
      int ci = A[i].Value; // C index
      UpdateC(C, ci, C[ci]+1);
   }

   //# step 6,7.
   for (int i = 1; i < C.Length; i++)
   {
      // C[i] = C[i] + C[i-1];
      UpdateC(C, i, C[i] + C[i - 1]);
   }

   //# step 8,9,10.
   AValueEx.ResetDataCanvas(); // for visualization

   for (int i = A.Length - 1; i >= 0; i--)
   {
      //# B[C[A[j]]] <- A[j]
     int di = C[A[i].Value] - 1; // zero base array.
      Update(datas, di, A[i]); // //datas[di] = A[i];  // 'datas' as B

      //# C[A[j]] <- C[A[j]] - 1
      int ci = A[i].Value;
      C[ci]--;

     AValueEx.DrawC(ci, C[ci]); // for visualization 變化太微量，看不出變化
   }
}
```

```
/// 基數排序法
public static void RadixSort(AValue[] datas)
{
   //## parameters
   const int BASE = 10;
   int m = 99; // datas.Max(); //## get max value

   // with bucket sort
   int n = datas.Length;
   AValue[] B = new AValue[n]; // as sorted datas.
   int exp = 1;
   while (m / exp > 0)
   {
      //## bucket sort
      int[] bucket = new int[BASE];

      AValueEx.ResetCountCanvas(); // for visualization

      for (int i = 0; i < n; i++)
      {
         //# bucket[(datas[i] / exp) % BASE]++;
         int bx = (datas[i].Value / exp) % BASE; // bucket index.
         UpdateC(bucket, bx, bucket[bx] + 1);  //bucket[bx]++;
      }

      for (int i = 1; i < BASE; i++)
      {
         //bucket[i] += bucket[i-1];
         UpdateC(bucket, i, bucket[i] + bucket[i-1]);
      }

     AValueEx.ResetDataCanvas(); // for visualization, //# put to calculated index

      for (int i = n - 1; i >= 0; i--)
      {
         // b[--bucket[(datas[i] / exp) % BASE]] = datas[i];
         int bx = (datas[i].Value / exp) % BASE;
         int bi = --bucket[bx];
         //B[bi] = datas[i];
         Update(B, bi, datas[i]);

         AValueEx.DrawC(bx, bucket[bx]); // for visualization, 變化太微量，看不出變化
      }

     AValueEx.CopyArray(B, 0, n - 1, datas); //// copy back for next round sorting

      // next round.
      exp *= BASE;
   }
}
```