資料結構作業四 - Sorting Algorithm

|  |  |  |  |
| --- | --- | --- | --- |
| 科系 | 學號 | 姓名 | 日期 |
| 電機系大二 | E24099059 | 陳旭祺 | 2021/06/20 |

* 程式

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  #include <time.h>  clock\_t begin**,** end**;**  double cost**;**  void insertionSort**(**int arr**[],** int n**)**  {  int i**,** key**,** j**;**  **for** **(**i **=** 1**;** i **<** n**;** i**++)** **{**  key **=** arr**[**i**];**  j **=** i **-** 1**;**  /\* Move elements of arr[0..i-1], that are  greater than key, to one position ahead  of their current position \*/  **while** **(**j **>=** 0 **&&** arr**[**j**]** **>** key**)** **{**  arr**[**j **+** 1**]** **=** arr**[**j**];**  j **=** j **-** 1**;**  **}**  arr**[**j **+** 1**]** **=** key**;**  **}**  **}** |
| void swap**(**int **\***xp**,** int **\***yp**)**  {  int temp **=** **\***xp**;**  **\***xp **=** **\***yp**;**  **\***yp **=** temp**;**  }  void selectionSort**(**int arr**[],** int n**){**  int i**,** j**,** min\_idx**;**  // One by one move boundary of unsorted subarray  **for** **(**i **=** 0**;** i **<** n**-**1**;** i**++)**  **{**  // Find the minimum element in unsorted array  min\_idx **=** i**;**  **for** **(**j **=** i**+**1**;** j **<** n**;** j**++)**  **if** **(**arr**[**j**]** **<** arr**[**min\_idx**])**  min\_idx **=** j**;**  // Swap the found minimum element with the first element  swap**(&**arr**[**min\_idx**],** **&**arr**[**i**]);**  **}**  **}** |
| // function to find the partition position  int partition**(**int array**[],** int low**,** int high**)** **{**  // select the rightmost element as pivot  int pivot **=** array**[**high**];**  // pointer for greater element  int i **=** **(**low **-** 1**);**  // traverse each element of the array  // compare them with the pivot  **for** **(**int j **=** low**;** j **<** high**;** j**++)** **{**  **if** **(**array**[**j**]** **<=** pivot**)** **{**  // if element smaller than pivot is found  // swap it with the greater element pointed by i  i**++;**  // swap element at i with element at j  swap**(&**array**[**i**],** **&**array**[**j**]);**  **}**  **}**  // swap the pivot element with the greater element at i  swap**(&**array**[**i **+** 1**],** **&**array**[**high**]);**  // return the partition point  return (i + 1);  }  void quickSort**(**int array**[],** int low**,** int high**)** **{**  **if** **(**low **<** high**)** **{**  // find the pivot element such that  // elements smaller than pivot are on left of pivot  // elements greater than pivot are on right of pivot  int pi **=** partition**(**array**,** low**,** high**);**  // recursive call on the left of pivot  quickSort**(**array**,** low**,** pi **-** 1**);**  // recursive call on the right of pivot  quickSort**(**array**,** pi **+** 1**,** high**);**  **}**  **}** |
| // Merges two subarrays of arr[].  // First subarray is arr[l..m]  // Second subarray is arr[m+1..r]  void merge**(**int arr**[],** int l**,** int m**,** int r**)**  {  int i**,** j**,** k**;**  int n1 **=** m **-** l **+** 1**;**  int n2 **=** r **-** m**;**  /\* create temp arrays \*/  int L**[**n1**],** R**[**n2**];**  /\* Copy data to temp arrays L[] and R[] \*/  **for** **(**i **=** 0**;** i **<** n1**;** i**++)**  L**[**i**]** **=** arr**[**l **+** i**];**  **for** **(**j **=** 0**;** j **<** n2**;** j**++)**  R**[**j**]** **=** arr**[**m **+** 1 **+** j**];**  /\* Merge the temp arrays back into arr[l..r]\*/  i **=** 0**;** // Initial index of first subarray  j **=** 0**;** // Initial index of second subarray  k **=** l**;** // Initial index of merged subarray  **while** **(**i **<** n1 **&&** j **<** n2**)** **{**  if (L[i] <= R[j]) {  arr**[**k**]** **=** L**[**i**];**  i**++;**  **}**  else {  arr**[**k**]** **=** R**[**j**];**  j**++;**  **}**  k**++;**  **}**  /\* Copy the remaining elements of L[], if there  are any \*/  while (i < n1) {  arr**[**k**]** **=** L**[**i**];**  i**++;**  k**++;**  **}**  /\* Copy the remaining elements of R[], if there  are any \*/  while (j < n2) {  arr**[**k**]** **=** R**[**j**];**  j**++;**  k**++;**  **}**  }  /\* l is for left index and r is right index of the  sub-array of arr to be sorted \*/  void mergeSort**(**int arr**[],** int l**,** int r**)**  {  **if** **(**l **<** r**)** **{**  // Same as (l+r)/2, but avoids overflow for  // large l and h  int m **=** l **+** **(**r **-** l**)** **/** 2**;**  // Sort first and second halves  mergeSort**(**arr**,** l**,** m**);**  mergeSort**(**arr**,** m **+** 1**,** r**);**  merge**(**arr**,** l**,** m**,** r**);**  **}**  **}** |
| void heapify**(**int arr**[],** int n**,** int i**)** **{**  // Find largest among root, left child and right child  int largest **=** i**;**  int left **=** 2 **\*** i **+** 1**;**  int right **=** 2 **\*** i **+** 2**;**  **if** **(**left **<** n **&&** arr**[**left**]** **>** arr**[**largest**])**  largest **=** left**;**  **if** **(**right **<** n **&&** arr**[**right**]** **>** arr**[**largest**])**  largest **=** right**;**  // Swap and continue heapifying if root is not largest  **if** **(**largest **!=** i**)** **{**  swap**(&**arr**[**i**],** **&**arr**[**largest**]);**  heapify**(**arr**,** n**,** largest**);**  **}**  **}**  // Main function to do heap sort  void heapSort**(**int arr**[],** int n**)** **{**  // Build max heap  **for** **(**int i **=** n **/** 2 **-** 1**;** i **>=** 0**;** i**--)**  heapify**(**arr**,** n**,** i**);**  // Heap sort  **for** **(**int i **=** n **-** 1**;** i **>=** 0**;** i**--)** **{**  swap**(&**arr**[**0**],** **&**arr**[**i**]);**  // Heapify root element to get highest element at root again  heapify**(**arr**,** i**,** 0**);**  **}**  **}** |
| // Function to get the largest element from an array  int getMax**(**int array**[],** int n**)** **{**  int max **=** array**[**0**];**  **for** **(**int i **=** 1**;** i **<** n**;** i**++)**  **if** **(**array**[**i**]** **>** max**)**  max **=** array**[**i**];**  return max;  }  // Using counting sort to sort the elements in the basis of significant places  void countingSort**(**int array**[],** int size**,** int place**)** **{**  int output**[**size **+** 1**];**  int max **=** **(**array**[**0**]** **/** place**)** **%** 10**;**  **for** **(**int i **=** 1**;** i **<** size**;** i**++)** **{**  **if** **(((**array**[**i**]** **/** place**)** **%** 10**)** **>** max**)**  max **=** array**[**i**];**  **}**  int count**[**max **+** 1**];**  **for** **(**int i **=** 0**;** i **<** max**;** **++**i**)**  count**[**i**]** **=** 0**;**  // Calculate count of elements  **for** **(**int i **=** 0**;** i **<** size**;** i**++)**  count**[(**array**[**i**]** **/** place**)** **%** 10**]++;**  // Calculate cumulative count  **for** **(**int i **=** 1**;** i **<** 10**;** i**++)**  count**[**i**]** **+=** count**[**i **-** 1**];**  // Place the elements in sorted order  **for** **(**int i **=** size **-** 1**;** i **>=** 0**;** i**--)** **{**  output**[**count**[(**array**[**i**]** **/** place**)** **%** 10**]** **-** 1**]** **=** array**[**i**];**  count**[(**array**[**i**]** **/** place**)** **%** 10**]--;**  **}**  **for** **(**int i **=** 0**;** i **<** size**;** i**++)**  array**[**i**]** **=** output**[**i**];**  **}**  // Main function to implement radix sort  void radixsort**(**int array**[],** int size**)** **{**  // Get maximum element  int max **=** getMax**(**array**,** size**);**  // Apply counting sort to sort elements based on place value.  **for** **(**int place **=** 1**;** max **/** place **>** 0**;** place **\*=** 10**)**  countingSort**(**array**,** size**,** place**);**  **}** |
| int main**(){**  int count**,**count2**;**  **for(**count **=** 10000**;** count **<=** 100000**;** count **=** count**+**10000**){**  printf**(**"\ncount = %d\n"**,**count**);**  srand**(**time**(NULL));**  int arr**[**count**],** arr2**[**count**];;**  int i**;**  int size **=** **sizeof(**arr**)** **/** **sizeof(**arr**[**0**]);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr**[**i**]** **=** rand**()** **%** 10000 **+** 1**;**  **}**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  insertionSort**(**arr2**,** count2**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Insertion sort time is: %lf secs\n"**,** cost**);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  selectionSort**(**arr2**,** count2**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Selection sort time is: %lf secs\n"**,** cost**);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  quickSort**(**arr2**,** 0**,** count2 **-** 1**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Quick sort time is: %lf secs\n"**,** cost**);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  mergeSort**(**arr2**,** 0**,** count2 **-** 1**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Merge sort time is: %lf secs\n"**,** cost**);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  heapSort**(**arr2**,** count2**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Heap sort time is: %lf secs\n"**,** cost**);**  **for(**i **=** 0**;** i **<** size**;** i**++){**  arr2**[**i**]** **=** arr**[**i**];**  **}**  count2 **=** count**;**  begin **=** clock**();**  radixsort**(**arr2**,** count2**);**  end **=** clock**();**  cost **=** **(**double**)(**end **-** begin**)/**CLOCKS\_PER\_SEC**;**  printf**(**"Radix sort time is: %lf secs\n"**,** cost**);**  **}**  return 0;  **}** |

* 結果

|  |
| --- |
| 一張含有 文字, 室外 的圖片  自動產生的描述 |

* 作圖分析

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm/Data size | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 |
| Insertion sort | 0.044 | 0.183 | 0.417 | 0.724 | 1.15 | 1.628 |
| Selection sort | 0.069 | 0.268 | 0.598 | 1.075 | 1.672 | 2.392 |
| Quick sort | 0.001 | 0.003 | 0.003 | 0.004 | 0.005 | 0.006 |
| Merge sort | 0.001 | 0.003 | 0.003 | 0.005 | 0.007 | 0.007 |
| Heap sort | 0.002 | 0.003 | 0.005 | 0.007 | 0.009 | 0.01 |
| Radix Sort | 0.001 | 0.002 | 0.002 | 0.004 | 0.005 | 0.005 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm/Data size | 7000 | 8000 | 9000 | 10000 | 150000 |
| Insertion sort | 2.19 | 2.866 | 3.655 | 4.547 | 10.561 |
| Selection sort | 3.263 | 4.275 | 5.341 | 6.61 | 20.055 |
| Quick sort | 0.008 | 0.008 | 0.009 | 0.012 | 0.018 |
| Merge sort | 0.01 | 0.01 | 0.012 | 0.014 | 0.02 |
| Heap sort | 0.012 | 0.015 | 0.017 | 0.018 | 0.028 |
| Radix Sort | 0.006 | 0.008 | 0.009 | 0.009 | 0.014 |

Unit: second (s)

* 分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Best time | Worst time | Avg time | Space | Stability |
| Insertion sort |  |  |  | 1 | Y |
| Selection sort |  |  |  | 1 | N |
| Quick sort |  |  |  |  | N |
| Merge sort |  |  |  |  | Y |
| Heap sort |  |  |  |  | N |
| Radix Sort |  |  |  |  | Y |

注:

：資料筆數。

：number of digit，資料中最多有幾個位數（或鍵值）

|  |
| --- |
| Insertion sort |
| 一張含有 文字, 電子用品, 鍵盤, 計算機 的圖片  自動產生的描述 |
| Selection sort |
| 一張含有 電子用品, 鍵盤, 計算機 的圖片  自動產生的描述 |
| Quick sort |
|  |
| Merge sort |
|  |
| Heap sort |
|  |

|  |
| --- |
| Radix Sort |
| 一張含有 文字, 丈量尺 的圖片  自動產生的描述 |
| Selection sort |
| 一張含有 電子用品, 鍵盤, 計算機 的圖片  自動產生的描述 |

* 心得

在網路上看到各式各樣的sorting，了解到各種sorting的優缺點與使用時機

像Insert sort雖然時間複雜度大但適用於資料量小或是已經幾乎快排好的序列。而Radix Sort採用分配的方式，依據多個鍵值排序，在某些時候比快速排序法要快，以我上面做的圖來看Radix Sort時間上小贏Quick Sort。

以下影片為將各種Sort視覺、聽覺化呈現，十分有趣。

[](https://www.youtube.com/embed/kPRA0W1kECg?feature=oembed)