第八章 排序

2022年10月2日 星期日 22:56

总结

名称	数据对象	稳定性	时间复杂度		额外空间复杂度	描述
			平均	最坏		
冒泡排序	数组	✓	$O(n^2)$		$O(1)$	（无序区，有序区）。 从无序区透过交换找出最大元素放到有序区前端。
选择排序	数组	✗	$O(n^2)$		$O(1)$	（有序区，无序区）。 在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。
	链表	✓				
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	（有序区，无序区）。 把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
堆排序	数组	✗	$O(n \log n)$		$O(1)$	（最大堆，有序区）。 从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组	✓	$O(n \log^2 n)$		$O(1)$	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。 可从上到下或从下到上进行。
			$O(n \log n)$		$O(n) + O(\log n)$ 如果不是从下到上	
	链表				$O(1)$	
快速排序	数组	✗	$O(n \log n)$	$O(n^2)$	$O(\log n)$	（小数，基准元素，大数）。 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。
希尔排序	数组	✗	$O(n \log^2 n)$	$O(n^2)$	$O(1)$	每一轮按照事先决定的间隔进行插入排序，间隔会依次缩小，最后一次一定要是1。
计数排序	数组、链表	✓	$O(n + m)$		$O(n + m)$	统计小于等于该元素值的元素的个数，于是该元素就放在目标数组的索引i位（i>0）。
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为i的元素放入i号桶，最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k \times n)$	$O(n^2)$		一种多关键字的排序算法，可用桶排序实现。

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	O(n²)	O(n)	O(n²)	O(1)	In-place	稳定
选择排序	O(n²)	O(n²)	O(n²)	O(1)	In-place	不稳定
插入排序	O(n²)	O(n)	O(n²)	O(1)	In-place	稳定
希尔排序	O(n log n)	O(n log² n)	O(n log² n)	O(1)	In-place	不稳定
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)	Out-place	稳定
快速排序	O(n log n)	O(n log n)	O(n²)	O(log n)	In-place	不稳定
堆排序	O(n log n)	O(n log n)	O(n log n)	O(1)	In-place	不稳定
计数排序	O(n + k)	O(n + k)	O(n + k)	O(k)	Out-place	稳定
桶排序	O(n + k)	O(n + k)	O(n²)	O(n + k)	Out-place	稳定
基数排序	O(n × k)	O(n × k)	O(n × k)	O(n + k)	Out-place	稳定

各种算法

主要看的是菜鸟教程的，可以直接看菜鸟教程里面的内容

1.冒泡排序

算法步骤

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

代码

```c
#include <stdio.h>
void bubble_sort(int arr[], int len)
{
    int i, j, temp;
    for (i = 0; i < len - 1; i++)
        for (j = 0; j < len - 1 - i; j++)
            if (arr[j] > arr[j + 1])
            {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
}
int main()
{
    int arr[] = {22, 34, 3, 32, 82, 55, 89, 50, 37, 5, 64, 35, 9, 70};
    int len = (int)sizeof(arr) / sizeof(*arr);
    bubble_sort(arr, len);
    int i;
    for (i = 0; i < len; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

动画图链接

https://www.runoob.com/w3cnote/bubble-sort.html

2.选择排序

- 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置。
- 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕。

```c
void swap(int *a, int *b) //交换两個變數
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
void selection_sort(int arr[], int len)
{
    int i, j;
    for (i = 0; i < len - 1; i++)
    {
        int min = i;
        for (j = i + 1; j < len; j++) //走訪未排序的元素
            if (arr[j] < arr[min])    //找到目前最小值
                min = j;              //紀錄最小值
        swap(&arr[min], &arr[i]);     //做交換
    }
}
```

动画图链接

https://www.runoob.com/w3cnote/selection-sort.html

3.插入排序

- 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
- 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

```c
void insertion_sort(int arr[], int len)
{
    int i, j, key;
    for (i = 1; i < len; i++)
    {
        key = arr[i];
        j = i - 1;
        while ((j >= 0) && (arr[j] > key))
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

动画图链接

https://www.runoob.com/w3cnote/insertion-sort.html

4.希尔排序

- 选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；
- 按增量序列个数 k，对序列进行 k 趟排序；
- 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```c
void shell_sort(int arr[], int len)
{
    int gap, i, j;
    int temp;
    for (gap = len >> 1; gap > 0; gap >>= 1)
        for (i = gap; i < len; i++)
        {
            temp = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > temp; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = temp;
        }
}
```

动画图链接

https://www.runoob.com/w3cnote/shell-sort.html

5.归并排序

- 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；
- 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
- 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
- 重复步骤 3 直到某一指针达到序列尾；
- 将另一序列剩下的所有元素直接复制到合并序列尾。

```c
void merge_sort_recursive(int arr[], int reg[], int start, int end)
{
    if (start >= end)
        return;
    int len = end - start, mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    merge_sort_recursive(arr, reg, start1, end1);
    merge_sort_recursive(arr, reg, start2, end2);
    int k = start;
    while (start1 <= end1 && start2 <= end2)
        reg[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];
    while (start1 <= end1)
        reg[k++] = arr[start1++];
    while (start2 <= end2)
        reg[k++] = arr[start2++];
    for (k = start; k <= end; k++)
        arr[k] = reg[k];
}
void merge_sort(int arr[], const int len)
{
    int reg[len];
    merge_sort_recursive(arr, reg, 0, len - 1);
}
```

动画图链接

https://www.runoob.com/w3cnote/merge-sort.html

6.快速排序

算法步骤

- 从数列中挑出一个元素，称为 "基准"（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

代码

```c
void swap(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}
void quick_sort_recursive(int arr[], int start, int end)
{
    if (start >= end)
        return;
    int mid = arr[end];
    int left = start, right = end - 1;
    while (left < right)
    {
        while (arr[left] < mid && left < right)
            left++;
        while (arr[right] >= mid && left < right)
            right--;
        swap(&arr[left], &arr[right]);
    }
    if (arr[left] >= arr[end])
        swap(&arr[left], &arr[end]);
    else
        left++;
    if (left)
        quick_sort_recursive(arr, start, left - 1);
    quick_sort_recursive(arr, left + 1, end);
}
void quick_sort(int arr[], int len)
{
    quick_sort_recursive(arr, 0, len - 1);
}
```

动画图链接

https://www.runoob.com/w3cnote/quick-sort-2.html

7.堆排序

- 创建一个堆 H[0……n-1]；
- 把堆首（最大值）和堆尾互换；
- 把堆的尺寸缩小 1，并调用 shift_down(0)，目的是把新的数组顶端数据调整到相应位置；
- 重复步骤 2，直到堆的尺寸为 1。

```c
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b)
{
    int temp = *b;
    *b = *a;
    *a = temp;
}
void max_heapify(int arr[], int start, int end)
{
    // 建立父節點指標和子節點指標
    int dad = start;
    int son = dad * 2 + 1;
    while (son <= end)
    {                                                  // 若子節點指標在範圍內才做比較
        if (son + 1 <= end && arr[son] < arr[son + 1]) // 先比較兩個子節點大小，選擇最大的
            son++;
        if (arr[dad] > arr[son]) //如果父節點大於子節點代表調整完畢，直接跳出函數
            return;
        else
        { // 否則交換父子內容再繼續子節點和孫節點比較
            swap(&arr[dad], &arr[son]);
            dad = son;
            son = dad * 2 + 1;
        }
    }
}
void heap_sort(int arr[], int len)
{
    int i;
    // 初始化，i從最後一個父節點開始調整
    for (i = len / 2 - 1; i >= 0; i--)
        max_heapify(arr, i, len - 1);
    // 先將第一個元素和已排好元素前一位做交換，再重新調整，直到排序完畢
    for (i = len - 1; i > 0; i--)
    {
        swap(&arr[0], &arr[i]);
        max_heapify(arr, 0, i - 1);
    }
}
int main()
{
    int arr[] = {3, 5, 3, 0, 8, 6, 1, 5, 8, 6, 2, 4, 9, 4, 7, 0, 1, 8, 9, 7, 3, 1,
2, 5, 9, 7, 4, 0, 2, 6};
    int len = (int)sizeof(arr) / sizeof(*arr);
    heap_sort(arr, len);
    int i;
    for (i = 0; i < len; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

动画图链接

https://www.runoob.com/w3cnote/heap-sort.html

8.计数排序

- （1）找出待排序的数组中最大和最小的元素
- （2）统计数组中每个值为i的元素出现的次数，存入数组C的第i项
- （3）对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）
- （4）反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void print_arr(int *arr, int n)
{
    int i;
    printf("%d", arr[0]);
    for (i = 1; i < n; i++)
        printf(" %d", arr[i]);
    printf("\n");
}
void counting_sort(int *ini_arr, int *sorted_arr, int n)
{
    int *count_arr = (int *)malloc(sizeof(int) * 100);
    int i, j, k;
    for (k = 0; k < 100; k++)
        count_arr[k] = 0;
    for (i = 0; i < n; i++)
        count_arr[ini_arr[i]]++;
    for (k = 1; k < 100; k++)
        count_arr[k] += count_arr[k - 1];
    for (j = n; j > 0; j--)
        sorted_arr[--count_arr[ini_arr[j - 1]]] = ini_arr[j - 1];
    free(count_arr);
}
int main(int argc, char **argv)
{
    int n = 10;
    int i;
    int *arr = (int *)malloc(sizeof(int) * n);
    int *sorted_arr = (int *)malloc(sizeof(int) * n);
    srand(time(0));
    for (i = 0; i < n; i++)
        arr[i] = rand() % 100;
    printf("ini_array: ");
    print_arr(arr, n);
    counting_sort(arr, sorted_arr, n);
    printf("sorted_array: ");
    print_arr(sorted_arr, n);
    free(arr);
    free(sorted_arr);
    return 0;
}
```

动画图链接

https://www.runoob.com/w3cnote/counting-sort.html

9.桶排序

- 桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。为了使桶排序更加高效，我们需要做到这两点：
- 在额外空间充足的情况下，尽量增大桶的数量
- 使用的映射函数能够将输入的 N 个数据均匀的分配到 K 个桶中

```c
#include <iterator>
#include <iostream>
#include <vector>
using namespace std;
const int BUCKET_NUM = 10;
struct ListNode
{
    explicit ListNode(int i = 0) : mData(i), mNext(NULL) {}
    ListNode *mNext;
    int mData;
};
ListNode *insert(ListNode *head, int val)
{
    ListNode dummyNode;
    ListNode *newNode = new ListNode(val);
    ListNode *pre, *curr;
    dummyNode.mNext = head;
    pre = &dummyNode;
    curr = head;
    while (NULL != curr && curr->mData <= val)
    {
        pre = curr;
        curr = curr->mNext;
    }
    newNode->mNext = curr;
    pre->mNext = newNode;
    return dummyNode.mNext;
}
ListNode *Merge(ListNode *head1, ListNode *head2)
{
    ListNode dummyNode;
    ListNode *dummy = &dummyNode;
    while (NULL != head1 && NULL != head2)
    {
        if (head1->mData <= head2->mData)
        {
            dummy->mNext = head1;
            head1 = head1->mNext;
        }
        else
        {
            dummy->mNext = head2;
            head2 = head2->mNext;
        }
        dummy = dummy->mNext;
    }
    if (NULL != head1)
        dummy->mNext = head1;
    if (NULL != head2)
        dummy->mNext = head2;
    return dummyNode.mNext;
}
void BucketSort(int n, int arr[])
{
    vector<ListNode *> buckets(BUCKET_NUM, (ListNode *)(0));
    for (int i = 0; i < n; ++i)
    {
        int index = arr[i] / BUCKET_NUM;
        ListNode *head = buckets.at(index);
        buckets.at(index) = insert(head, arr[i]);
    }
    ListNode *head = buckets.at(0);
    for (int i = 1; i < BUCKET_NUM; ++i)
    {
        head = Merge(head, buckets.at(i));
    }
    for (int i = 0; i < n; ++i)
    {
        arr[i] = head->mData;
        head = head->mNext;
    }
}
```

动画图链接

https://www.runoob.com/w3cnote/bucket-sort.html

10.基数排序

- 1.10 基数排序
- 分类 算法
- 基数排序是一种非比较型整数排序算法，其原理是将整数按位数切割成不同的数字，然后按每个位数分别比较。由于整数也可以表达字符串（比如名字或日期）和特定格式的浮点数，所以基数排序也不是只能使用于整数。
- 基数排序：根据键值的每位数字来分配桶；
- 计数排序：每个桶只存储单一键值；
- 桶排序：每个桶存储一定范围的数值；

```c
#include <stdio.h>
#define MAX 20
//#define SHOWPASS
#define BASE 10
void print(int *a, int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        printf("%d\t", a[i]);
    }
}
void radixsort(int *a, int n)
{
    int i, b[MAX], m = a[0], exp = 1;
    for (i = 1; i < n; i++)
    {
        if (a[i] > m)
        {
            m = a[i];
        }
    }
    while (m / exp > 0)
    {
        int bucket[BASE] = {0};
        for (i = 0; i < n; i++)
        {
            bucket[(a[i] / exp) % BASE]++;
        }
        for (i = 1; i < BASE; i++)
        {
            bucket[i] += bucket[i - 1];
        }
        for (i = n - 1; i >= 0; i--)
        {
            b[--bucket[(a[i] / exp) % BASE]] = a[i];
        }
        for (i = 0; i < n; i++)
        {
            a[i] = b[i];
        }
        exp *= BASE;
#ifdef SHOWPASS
        printf("\nPASS   : ");
        print(a, n);
#endif
    }
}
int main()
{
    int arr[MAX];
    int i, n;
    printf("Enter total elements (n <= %d) : ", MAX);
    scanf("%d", &n);
    n = n < MAX ? n : MAX;
    printf("Enter %d Elements : ", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    printf("\nARRAY  : ");
    print(&arr[0], n);
    radixsort(&arr[0], n);
    printf("\nSORTED : ");
    print(&arr[0], n);
    printf("\n");
    return 0;
}
```

动画图链接

https://www.runoob.com/w3cnote/radix-sort.html