十大排序算法

2021年7月28日 9:53

十大排序算法（https://blog.csdn.net/lxk2017/article/details/100191120）：

1、冒泡排序（Bubble Sort）
2、选择排序（Selection Sort）
3、插入排序（Insertion Sort）
4、希尔排序（Shell Sort）
5、归并排序（Merge Sort）
6、快速排序（Quick Sort）
7、堆排序（Heap Sort）
8、计数排序（Counting Sort）
9、桶排序（Bucket Sort）
10、基数排序（Radix Sort）

排序算法相关说明：

定义：

对一序列对象根据某种顺序关系进行排序。

术语说明：

（1）稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
（2）不稳定：如果a原本在b前面，而a=b，排序之后a可能会出现在b的后面；
（3）内排序：所有排序操作都在内存中完成；
（4）外排序：由于数据天大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
（5）时间复杂度：一个算法执行所耗费的时间；
（6）空间复杂度：运行完一个程序所需内存的大小。

复杂度与稳定性：

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	$O(n^2)$	O(n)	$O(n^2)$	O(1)	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	In-place	不稳定
插入排序	$O(n^2)$	O(n)	$O(n^2)$	O(1)	In-place	稳定
希尔排序	O(n log n)	$O(n \log^2 n)$	$O(n \log^2 n)$	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)	$O(n^2)$	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)	$O(n^2)$	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

说明：

n：数据规模；
k：“桶”的个数；
In-place：占用常数内存，不占额外内存；
Out-place：占用额外内存。

排序算法分类：

1、冒泡排序

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

1.1算法描述

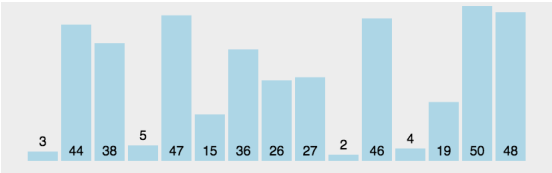
（1）比较相邻的元素，如果第一个比第二个大，就交换它们两个；

（2）对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

（3）针对所有的元素重复以上的步骤，除了最后一个；

（4）重复步骤（1）~（3），直到排序完成。

1. 2动态演示



1. 3代码实现

```
vector<int> bubble (vector<int>intput_vector)
{
    if(intput_vector.size()==0)
        {
            return intput_vector;
        }
    for (size_t i = 0; i < intput_vector.size(); i++)
    {
        for (size_t j = 0; j < intput_vector.size()-1-i; j++)
        {
            if (intput_vector[j]>intput_vector[j + 1])
            {
                int temp = intput_vector[j];
                intput_vector[j] = intput_vector[j + 1];
                intput_vector[j + 1] = temp;
            }
        }
    }
    return intput_vector;
}
```

1. 4算法分析

最佳情况：T(n) = O(n) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)。

2、选择排序

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：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，然后，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。以此类推，直到所有元素均排序完毕。

2.1算法描述

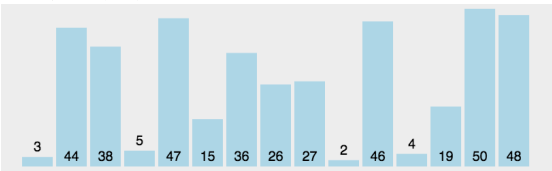
n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

（1）初始状态：无序区为R[1…n]，有序区为空；

（2）第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1…i-1]和R(i…n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1…i]和R[i+1…n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；

（3）n-1趟结束，数组有序化了。

2. 2动态演示



2. 3代码实现

```cpp
vector<int> selectionSort (vector<int>intput_vector)
{
    if (intput_vector.size()==0)
    {
        return intput_vector;
    }
    for (size_t i = 0; i < intput_vector.size(); i++)
    {
        int minIndex = i;
        for (size_t j = i; j < intput_vector.size(); j++)
        {
            if (intput_vector[j]<intput_vector[minIndex])
            {
                minIndex = j;
            }
        }
        int temp = intput_vector[minIndex];
        intput_vector[minIndex] = intput_vector[i];
        intput_vector[i] = temp;
    }
    return intput_vector;
}
```

2.4算法分析

最佳情况：T(n) = O(n2) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n2)。

3、插入排序

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

3.1算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

（1）从第一个元素开始，该元素可以认为已经被排序；

（2）取出下一个元素，在已经排序的元素序列中从后向前扫描；

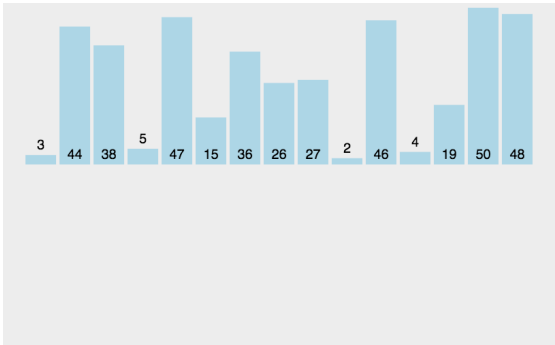
（3）如果该元素（已排序）大于新元素，将该元素移到下一位置；

（4）重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

（5）将新元素插入到该位置后；

（6）重复步骤2~5。

3.2动态演示



3.3代码实现

```cpp
vector<int> insertionSort (vector<int>intput_vector)
{
    if (intput_vector.size()==0)
    {
        return intput_vector;
    }
    int current ;
    for (size_t i = 0; i < intput_vector.size()-1; i++)
    {
        current = intput_vector[i + 1];//因为i+1所以i<...size()-1
        int preIndex = i;
        while (preIndex>=0 && current<intput_vector[preIndex])
        {
            intput_vector[preIndex + 1] = intput_vector[preIndex];
            preIndex--;
        }
        intput_vector[preIndex + 1] = current;
    }
    return intput_vector;
}
```

3.4算法分析

最佳情况：T(n) = O(n) 最坏情况：T(n) = O(n2) 平均情况：T(n) = O(n2)。

4、希尔排序

希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

希尔排序是把记录按下表的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

4.1算法描述

我们来看下希尔排序的基本步骤，在此我们选择增量gap=length/2，缩小增量继续以gap = gap/2的方式，这种增量选择我们可以用一个序列来表示，{n/2,(n/2)/2...1}，称为增量序列。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。此处我们做示例使用希尔增量。

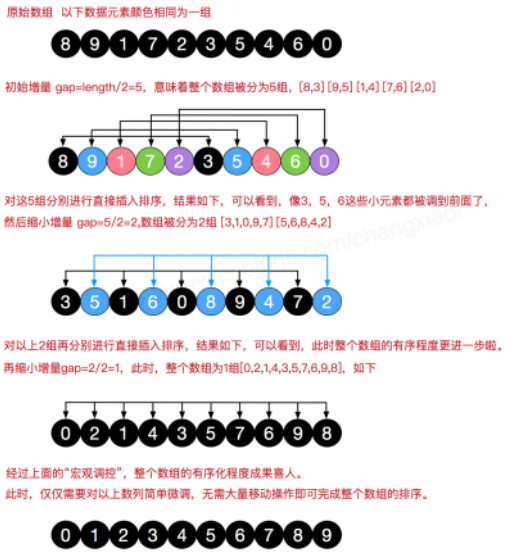
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

（1）选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

（2）按增量序列个数k，对序列进行k 趟排序；

（3）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。

4.2过程演示



4.3代码实现

```
vector<int> ShellSort (vector<int>intput_vector)
{
    if (intput_vector.size()==0)
    {
        return intput_vector;
    }
    int len=intput_vector.size() ;
    int temp, gap = len / 2;//算法采用希尔增量
    while (gap>0)
    {
        for (size_t i = gap; i < len; i++)
        {
            temp = intput_vector[i];
            int preIndex = i - gap;
            while (preIndex>=0 && intput_vector[preIndex]>temp)
            {
                intput_vector[preIndex + gap] = intput_vector[preIndex];
                preIndex -= gap;
            }
            intput_vector[preIndex + gap] = temp;
        }
        gap /= 2;
    }
    return intput_vector;
}
```

4.4算法分析

最佳情况：T(n) = O(nlog2 n) 最坏情况：T(n) = O(nlog2 n) 平均情况：T(n) =O(nlog2n)。

5、归并排序

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

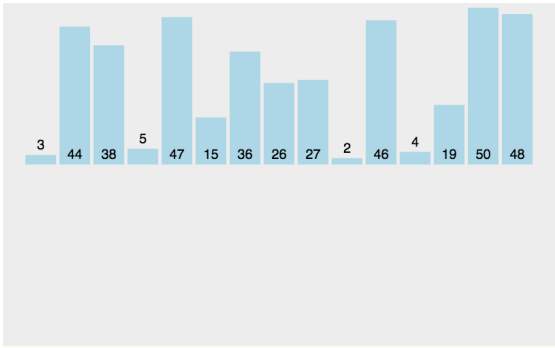
5.1算法描述

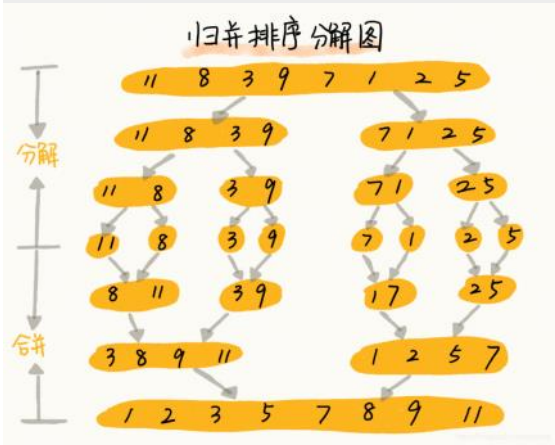
（1）把长度为n的输入序列分成两个长度为n/2的子序列；

（2）对这两个子序列分别采用归并排序；

（3）将两个排序好的子序列合并成一个最终的排序序列。

5.2动态演示





5.3代码实现

```
//二路归并排序算法
vector<int> MergeSort(vector<int>input_vetror)
{
    if (input_vetror.size()<2)
    {
        return input_vetror;
    }
    int mid = input_vetror.size()/ 2;
    vector<int>left_vector(input_vetror.begin(),input_vetror.begin()+mid);
    vector<int>right_vector(input_vetror.begin()+mid, input_vetror.end());
    return merge(MergeSort(left_vector), MergeSort(right_vector));
}
vector<int> merge(vector<int> left, vector<int>right)
{
    vector<int> reusult;
    for (size_t i = 0, j = 0, k = 0; (j < left.size() || k < right.size());)
    {
        if ((j < left.size()) && ((k < right.size()) || (left[j] <= right[k])))
        {
            reusult.push_back(left[j++]);
        }
        if ((k < right.size()) && ((j < left.size()) || (right[k] < left[j])))
        {
            reusult.push_back(right[k++]);
        }
    }
    return reusult;
}
```

5.4算法分析

最佳情况：T(n) = O(n) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)。

6、快速排序

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

6.1算法描述

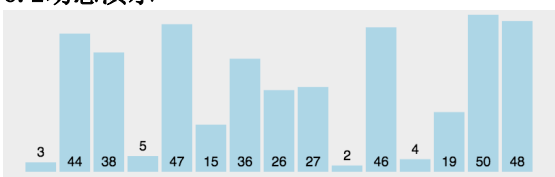
快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

（1）从数列中挑出一个元素，称为 “基准”（pivot）；

（2）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；

（3）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。

6.2动态演示



6.3代码实现

```
int partition(vector<int> &input_vector, size_t start, size_t end)
{
    swap(input_vector[start], input_vector[start + rand() % (end - start + 1)]);
```

```
    int pivot = input_vector[start];
    while (start<end)
    {
        while (start<end)
        {
            if (pivot < input_vector[end])
                end--;
            else
            {
                input_vector[start++] = input_vector[end];
                break;
            }
        }
        while (start<end)
        {
            if (input_vector[start]<pivot)
                start++;
            else
            {
                input_vector[end--] = input_vector[start];
                break;
            }
        }
    }
    input_vector[start] = pivot;
    return start;
}
vector<int> QuickSort(vector<int> &input_vetor, size_t start, size_t end)
{
    if (end-start<2) //递归基
    {
        return input_vetor;
    }
    size_t mi = partition(input_vetor, start, end);
    if (mi>start)
    {
        QuickSort(input_vetor, start, mi - 1);
    }
    if (mi<end)
    {
        QuickSort(input_vetor, mi, end);
    }
    return input_vetor;
}
```

6.4算法分析

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(nlogn)。

7、堆排序

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

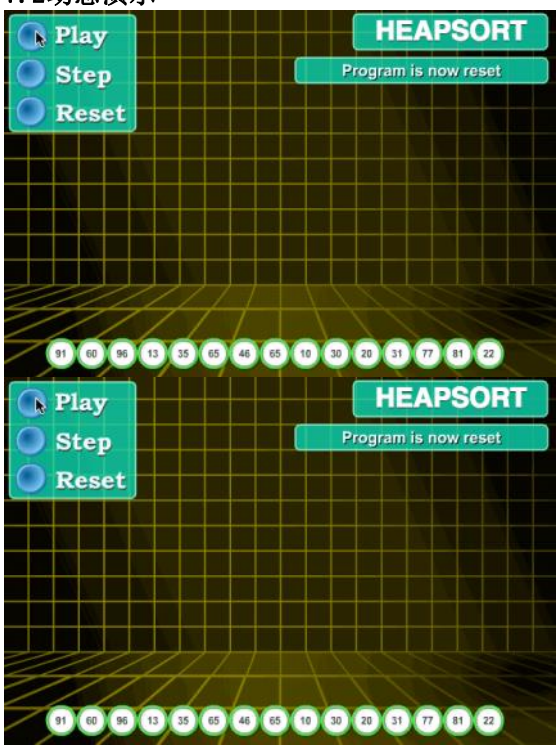
7.1算法描述

（1）将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

（2）将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

（3）由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

7.2动态演示



7.3代码实现

```
vector<int> HeapSort(vector<int> input_vector)
{
    len = input_vector.size();
    if (len < 1) return input_vector;
    //1、构建一个大顶堆
    buildMaxHeap(input_vector);
    while (len > 0)
    {
        swap(input_vector[0], input_vector[len-1]);
        len--;
        adjustHeap(input_vector, 0);
    }
    return input_vector;
}
void buildMaxHeap(vector<int> & input_vector)//Floyd快速建堆算法
{
    for (int i =len/2-1; i>=0; i--) //从堆的最后一个内部节点开始调整
    {
        adjustHeap(input_vector, i);
    }
}
void adjustHeap(vector<int>& input_vector, size_t i)//完成自上而下的过滤
{
    int maxIndex = i;
    //如果有左子树，且左子树大于父节点则将最大指针指向左子树（如果一个节点有左孩子则它的秩为：2*i(v)+1）
    if (i*2+1<len && input_vector[i*2+1]> input_vector[maxIndex])
        maxIndex = i * 2+1;
    //如果有右子树，且右子树大于父节点则将最大指针指向右子树（如果一个节点有右孩子则它的秩为：2*i(v)+2）
    if (i * 2 + 2 < len && input_vector[i * 2 + 2] > input_vector[maxIndex])
        maxIndex = i * 2 + 2;
    //基于堆序性如果父节点不是最大值，执行交换操作
    if (maxIndex!=i)
    {
        swap(input_vector[i], input_vector[maxIndex]);
        adjustHeap(input_vector, maxIndex);
    }
}
```

7.4算法分析

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)。

8、计数排序

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

8.1算法描述

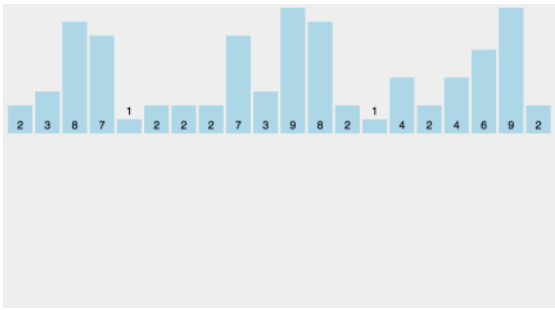
（1）找出待排序的数组中最大和最小的元素；

（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。

8.2动态演示



8.3代码实现

```
vector<int> CountingSort(vector<int>& input_vector,int maxVal)
{
    int len = input_vector.size();
    if (len < 1) return input_vector;
    vector<int> count(maxVal + 1, 0);
    vector<int>temp(input_vector);
    for (auto x:input_vector)//获取每个元素的个数在count中元素值为下标
        count[x]++;
    for (int i = 1; i <= maxVal; i++)
        count[i] += count[i - 1];
    for (int i = len-1; i>=0; i--)
    {
        input_vector[count[temp[i]] - 1] = temp[i];
        count[temp[i]]--;
    }
    return input_vector;
}
```

8.4算法分析

当输入的元素是n 个0到k之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n+k) 平均情况：T(n) = O(n+k)。

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。

桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排。

9.1算法描述

（1）人为设置一个BucketSize，作为每个桶所能放置多少个不同数值（例如当BucketSize==5时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放100个3）；

（2）遍历输入数据，并且把数据一个一个放到对应的桶里去；

（3）对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；

（4）从不是空的桶里把排好序的数据拼接起来。

注意，如果递归使用桶排序为各个桶排序，则当桶数量为1时要手动减小BucketSize增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。

9.2图片演示

9.3代码实现

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
void BucketSort(vector<int>& input_vector)
{
	int max = input_vector[0];
	int min = input_vector[0];
	for (auto it : input_vector) {
		if (it > max) max = it;
		if (it < min) min = it;
		int bucketNum = (max - min) / input_vector.size() + 1;
		vector<vector<int>> vecBucket;
		for (size_t i = 0; i < bucketNum; i++) {
			vector<int> vecTmp;
			vecBucket.push_back(vecTmp);
		}
		for (size_t i = 0; i < input_vector.size(); i++) {
			if (bucketNum == 1) {
				vecBucket[0].push_back(input_vector[i]);
			}
			else {
				int bucketIndex = (input_vector[i] - min) / bucketNum;
				bucketIndex = bucketIndex >= bucketNum ? bucketNum - 1  : bucketIndex;
				vecBucket[bucketIndex].push_back(input_vector[i]);
			}
		}
		input_vector.clear();
		for (vector<vector<int>>::iterator iter = vecBucket.begin(); iter !=  vecBucket.end(); iter++) {
			vector<int> vecTmp = *iter;
			if (vecTmp.size() <= 0) {
				continue;
			}
			sort(vecTmp.begin(), vecTmp.end());
			for (auto it : vecTmp) {
				input_vector.push_back(it);
			}
		}
	}
}
void main() {
	vector<int> v;
	v.push_back(5);
	v.push_back(1);
	v.push_back(2);
	v.push_back(4);
	v.push_back(6);
	v.push_back(10);
	v.push_back(3);
	v.push_back(6);
	v.push_back(0);
```

```
    v.push_back(9);
    BucketSort(v);
    for (auto item : v)
        cout << item << endl;
}
```

9.4算法分析

桶排序最好情况下使用线性时间O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

最佳情况：T(n) = O(n+k) 最差情况：T(n) = O(n2) 平均情况：T(n) = O(n+k)。

10、基数排序

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为O(kn)，n为数组长度，k为数组中的数的最大的位数；

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

10.1算法描述

（1）取得数组中的最大数，并取得位数；

（2）arr为原始数组，从最低位开始取每个位组成radix数组；

（3）对radix进行计数排序（利用计数排序适用于小范围数的特点）；

10.2动态演示

10.3代码实现

```
vector<int> RadixSort(vector<int>input_vector)
{
    if (input_vector.size() < 2) return input_vector;
    int max = input_vector[0];
    for (auto x : input_vector)
    {
        if (x > max) max = x;
    }
    int maxDigit = 0;
    while (max!=0)
    {
        max /= 10;
        maxDigit++;
    }
    int mod = 10, div = 1;
    vector<vector<int>> bucketList;
    for (size_t i = 0; i < 10; i++)
    {
        vector<int>avector;
        bucketList.push_back(avector);
    }
    for (int i = 0; i < maxDigit; i++, mod *= 10, div *= 10)
    {
        for (int j = 0; j < input_vector.size(); j++)
        {
            int num = (input_vector[j] % mod) / div;
            bucketList[num].push_back(input_vector[j]);
        }
        int index = 0;
        for (int  j = 0; j < bucketList.size(); j++)
        {
            for (int k = 0; k < bucketList[j].size(); k++)
                input_vector[index++] = bucketList[j][k];
            bucketList[j].clear();
        }
    }
    return input_vector;
}
```

10.4算法分析

最佳情况：T(n) = O(n * k) 最差情况：T(n) = O(n * k) 平均情况：T(n) = O(n * k)

基数排序 vs 计数排序 vs 桶排序，这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

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；

来自 <https://blog.csdn.net/lxk2017/article/details/100191120>