排序算法

Subtitle

2024/05/19

Table of Contents

排序算法

交换排序

冒泡排序

1. 依次比较相邻两数大小，大的后移
2. 重复此过程，共 len-1 趟
3. 已经排序好的不需要在排，因此趟数递增，需比较的数组长度递减
4. 算法复杂度为O(n^2) <file c bubble.cpp> #include <iostream> using namespace std;

void bubbleSort(int arr[],int len) {

```
int i,j,tmp;
for(i=0;i<len-1;i++)
{
   for(j=0;j<len-i-1;j++)
  {
      if(arr[j+1]<arr[j])
    {
         tmp=arr[j+1];
         arr[j+1]=arr[j];
         arr[j]=tmp;
    }
  }
   for(int k=0;k<len;k++)
      cout<<arr[k]<<" ";
   cout<<endl;
}
```

}

int main(int argc, char* argv[]) {

```
int arr[]={5,4,3,2,1};
int len=5;
for(int i=0;i<len;i++)
   cout<<arr[i]<<" ";
cout<<endl;
bubbleSort(arr,len);
return 0;
```

}

</file> 排序过程：

```
NPP_EXEC: "GCC_Run"
bubble.exe
Process started >>>
5 4 3 2 1
4 3 2 1 5
```

```
3 2 1 4 5
2 1 3 4 5
1 2 3 4 5
<<< Process finished. (Exit code 0)
```

快速排序

1. 从 a[start..end]中选取一个pivot元素，比如选a[start]
2. 从右至左搜索小于pivot的数，移到a[start]的位置上去，在从左至右找大于pivot的数，移到右边空缺的位置上
3. 递归

quickSort.cpp

```
#include <iostream>
using namespace std;

int partition(int start, int end, int arr[])
{
  //从a[start..end]中选取一个pivot元素（比如选a[start]为pivot);
  int pivot=arr[start];
  int l=start,r=end;

  while(l<r)
  {
    while(l<r && arr[r]>=pivot)
        r--;
    if(l<r)
    {
        arr[l]=arr[r];
        l++;
    }

    while(l<r && arr[l]<pivot)
        l++;
    if(l<r)
    {
        arr[r]=arr[l];
        r--;
    }

  }
   arr[l]=pivot;
  return l;
}

void quickSort(int start, int end, int arr[])
{
  int mid;
  if (end > start) {
      mid = partition(start, end ,arr);
      quickSort(start, mid-1, arr);
      quickSort(mid+1, end, arr);
```

```
    }
}

int main(int argc, char* argv[])
{
    int arr[]={10,5,4,2,3,1,8};
    int len=7;
     quickSort(0,len-1,arr);
    for(int i=0;i<len;i++)
        cout<<arr[i]<<" ";
    cout<<endl;
    return 0;
}
```

运行结果：

```
NPP_EXEC: "GCC_Run"
quickSort.exe
Process started >>>
1 2 3 4 5 8 10
<<< Process finished. (Exit code 0)
```

插入排序

直接插入排序

1. arr[0]是排好的
2. 从arr[1]开始，依次抽出一个数key
3. 将key插入到已排序数列中的合适位置

insertSort.cpp

```
#include <iostream>
using namespace std;
void insertSort(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;
      for(int k=0;k<len;k++)
         cout<<arr[k]<<" ";
      cout<<endl;
```

```
    }
}
int main(int argc, char* argv[])
{
    int arr[]={10, 5, 2, 4, 7};
    int len=5;
    for(int i=0;i<len;i++)
        cout<<arr[i]<<" ";
    cout<<endl;


     insertSort(arr,len);
    return 0;
}
```

运行结果

```
NPP_EXEC: "GCC_Run"
insertSort.exe
Process started >>>
10 5 2 4 7
5 10 2 4 7
2 5 10 4 7
2 4 5 10 7
2 4 5 7 10
<<< Process finished. (Exit code 0)
```

希尔排序

希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。(希尔排序)
原始的算法实现在最坏的情况下需要进行O(n2)的比较和交换。V. Pratt的书对算法进行了少量修改，可以使得性能提升至O(n $\log^2$ n)。这比最好的比较算法的O(n log n)要差一些。

shellSort.cpp

```
#include <iostream>
using namespace std;

void shellPass(int arr[],int len,int gap)
{
    int i,j,key;
    //步长不总为1的插入排序
    for ( i = gap; i < len; i++ )
        {
          j = i - gap;
          key = arr[i];
        while (( j >= 0 ) && ( arr[j] > key ))
        {
              arr[j + gap] = arr[j];
              j = j - gap;        //在已排序序列中寻找合适位置插入key
```

```
    }
      arr[j + gap] = key;
    for(int i=0;i<len;i++)
      cout<<arr[i]<<" ";
    cout<<endl;
    }
}

void shellSort(int arr[],int len)
{
  int gap=0;
  while(gap<=len)
      gap=gap*3+1;  //确保最终步长为1
  while(gap>0)
  {
      shellPass(arr,len,gap);
      gap=(gap-1)/3;
  }
}

int main(int argc, char* argv[])
{
  int arr[]={10,54,2,3,1,0,3,5};
  int len=8;
   shellSort(arr,len);
}
```

运行结果：

```
NPP_EXEC: "GCC_Run"
shellSort.exe
Process started >>>
1 54 2 3 10 0 3 5
1 0 2 3 10 54 3 5
1 0 2 3 10 54 3 5
1 0 2 3 10 54 3 5
0 1 2 3 10 54 3 5
0 1 2 3 10 54 3 5
0 1 2 3 10 54 3 5
0 1 2 3 10 54 3 5
0 1 2 3 10 54 3 5
0 1 2 3 3 10 54 5
0 1 2 3 3 5 10 54
<<< Process finished. (Exit code 0)
```

归并排序

1. Divide: 把长度为n的输入序列分成两个长度为n/2的子序列。
2. Conquer: 对这两个子序列分别采用归并排序。
3. Combine: 将两个排序好的子序列合并成一个最终的排序序列。

mergeSort.cpp

```
#include <iostream>
using namespace std;

void merge(int start,int mid,int end,int arr[])
{
  int i,j,k;
  int len1=mid-start+1;
  int len2=end-mid;
  int left[len1],right[len2];
  for(k=0;k<len1;k++)
      left[k]=arr[start+k];
  for(k=0;k<len2;k++)
      right[k]=arr[mid+1+k];
   k=start;
   i=j=0;
  while(i<len1 && j<len2)
  {
    if(left[i]<right[j])
        arr[k++]=left[i++];
    else
        arr[k++]=right[j++];
  }

  while(i<len1)
      arr[k++]=left[i++];
  while(j<len2)
      arr[k++]=right[j++];


}

void sort(int start,int end,int arr[])
{
  int mid;
  if(start<end)
  {
      mid=(start+end)/2;
      sort(start,mid,arr);
      sort(mid+1,end,arr);
      merge(start,mid,end,arr);
  }

}

void mergeSort(int arr[],int len)
{
   sort(0,len-1,arr);
}

int main(int argc, char* argv[])
{
  int arr[]={5,4,3,2};
  int len=4;
```

```
    mergeSort(arr,len);
   for(int i=0;i<len;i++)
      cout<<arr[i]<<" ";
   cout<<endl;
   return 0;
}
```

sort函数把a[start..end]平均分成两个子序列，分别是a[start..mid]和a[mid+1..end]，对这两个子序列分别递归调用sort函数进行排序，然后调用merge函数将排好序的两个子序列合并起来，由于两个子序列都已经排好序了，合并的过程很简单，每次循环取两个子序列中最小的元素进行比较，将较小的元素取出放到最终的排序序列中，如果其中一个子序列的元素已取完，就把另一个子序列剩下的元素都放到最终的排序序列中。

执行结果

```
NPP_EXEC: "GCC_Run"
mergeSort.exe
Process started >>>
2 3 4 5
<<< Process finished. (Exit code 0)
```

堆排序

参见 堆排序

heapSort.cpp

```
#include <cstdio>
#include <cstdlib>
#include <cmath>
using namespace std;

int parent(int);
int left(int);
int right(int);
void Max_Heapify(int [], int, int);
void Build_Max_Heap(int [], int);
void print(int [], int);
void HeapSort(int [], int);

/*父結點*/
int parent(int i)
{
   return (int)floor((i - 1) / 2);
}

/*左子結點*/
int left(int i)
{
   return (2 * i + 1);
}
```

```
/*右子結點*/
int right(int i)
{
  return (2 * i + 2);
}

/*單一子結點最大堆積樹調整*/
void Max_Heapify(int A[], int i, int heap_size)
{
  int l = left(i);
  int r = right(i);
  int largest;
  int temp;
  if(l < heap_size && A[l] > A[i])
  {
      largest = l;
  }
  else
  {
      largest = i;
  }
  if(r < heap_size && A[r] > A[largest])
  {
      largest = r;
  }
  if(largest != i)
  {
      temp = A[i];
      A[i] = A[largest];
      A[largest] = temp;
      Max_Heapify(A, largest, heap_size);
  }
}

/*建立最大堆積樹*/
void Build_Max_Heap(int A[],int heap_size)
{
  for(int i = (heap_size-2)/2; i >= 0; i--)
  {
      Max_Heapify(A, i, heap_size);
  }
}

/*印出樹狀結構*/
void print(int A[], int heap_size)
{
  for(int i = 0; i < heap_size;i++)
  {
    printf("%d ", A[i]);
  }
  printf("\n");
}

/*堆積排序程序碼*/
```

```
void HeapSort(int A[], int heap_size)
{
    Build_Max_Heap(A, heap_size);
  int temp;
  for(int i = heap_size - 1; i >= 0; i--)
  {
      temp = A[0];
      A[0] = A[i];
      A[i] = temp;
      Max_Heapify(A, 0, i);
  }
    print(A, heap_size);
}

/*輸入資料並做堆積排序*/
int main(int argc, char* argv[])
{
  const int heap_size = 13;
  int A[] = {19, 1, 10, 14, 16, 4, 7, 9, 3, 2, 8, 5, 11};
    HeapSort(A, heap_size);
  system("pause");
  return 0;
}
```

排序算法时间比较

测试代码

sortTest.cpp

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "sort.h"
using namespace std;

void randArr(int *arr,int len)
{
  int i;
  srand((int)time(0));
  for(i=0;i<len;i++)
  {
      arr[i]=rand()%len;
  }
}
int main(int argc, char* argv[])
{
  int len;
  cout<<"len: ";
  cin>>len;
  int *arr=new int[len];
```

```
clock_t start,finish;
double total;
//冒泡
 randArr(arr,len);
 start=clock();
 bubbleSort(arr,len);
 finish=clock();
 total=(int)(finish-start);
cout<<"Bubble: "<<total<<" ms"<<endl;

//插入
 randArr(arr,len);
 start=clock();
 insertSort(arr,len);
 finish=clock();
 total=(int)(finish-start);
cout<<"Insert: "<<total<<" ms"<<endl;

//归并
 randArr(arr,len);
 start=clock();
 sort(0,len-1,arr);
 finish=clock();
 total=(int)(finish-start);
cout<<"Merge: "<<total<<" ms"<<endl;

//快速
 randArr(arr,len);
 start=clock();
 quickSort(0,len-1,arr);
 finish=clock();
 total=(int)(finish-start);
cout<<"Quick: "<<total<<" ms"<<endl;

//希尔
 randArr(arr,len);
 start=clock();
 shellSort(arr,len);
 finish=clock();
 total=(int)(finish-start);
cout<<"Shell: "<<total<<" ms"<<endl;

//堆
 randArr(arr,len);
 start=clock();
 HeapSort(arr,len);
 finish=clock();
 total=(int)(finish-start);
cout<<"Heap: "<<total<<" ms"<<endl;

delete [] arr;
```

```
  return 0;
}
```

头文件

sort.h

```
#include <cstdio>
#include <cstdlib>
#include <cmath>

//Bubble
void bubbleSort(int arr[],int len)
{
  int i,j,tmp;
  for(i=0;i<len-1;i++)
  {
    for(j=0;j<len-i-1;j++)
    {
      if(arr[j+1]<arr[j])
      {
            tmp=arr[j+1];
            arr[j+1]=arr[j];
            arr[j]=tmp;
      }

    }
  }

}

//Insert
void insertSort(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;
  }
}

//Merge
void merge(int start,int mid,int end,int arr[])
{
  int i,j,k;
```

```
  int len1=mid-start+1;
  int len2=end-mid;
  int left[len1],right[len2];
  for(k=0;k<len1;k++)
      left[k]=arr[start+k];
  for(k=0;k<len2;k++)
      right[k]=arr[mid+1+k];
   k=start;
   i=j=0;
  while(i<len1 && j<len2)
  {
    if(left[i]<right[j])
        arr[k++]=left[i++];
    else
        arr[k++]=right[j++];
  }

  while(i<len1)
      arr[k++]=left[i++];
  while(j<len2)
      arr[k++]=right[j++];


}

void sort(int start,int end,int arr[])
{
  int mid;
  if(start<end)
  {
      mid=(start+end)/2;
      sort(start,mid,arr);
      sort(mid+1,end,arr);
      merge(start,mid,end,arr);
  }

}

void mergeSort(int arr[],int len)
{
   sort(0,len-1,arr);
}

//Quick
int partition(int start, int end, int arr[])
{
  //从a[start..end]中选取一个pivot元素（比如选a[start]为pivot□;
  int pivot=arr[start];
  int l=start,r=end;

  while(l<r)
  {
    while(l<r && arr[r]>=pivot)
```

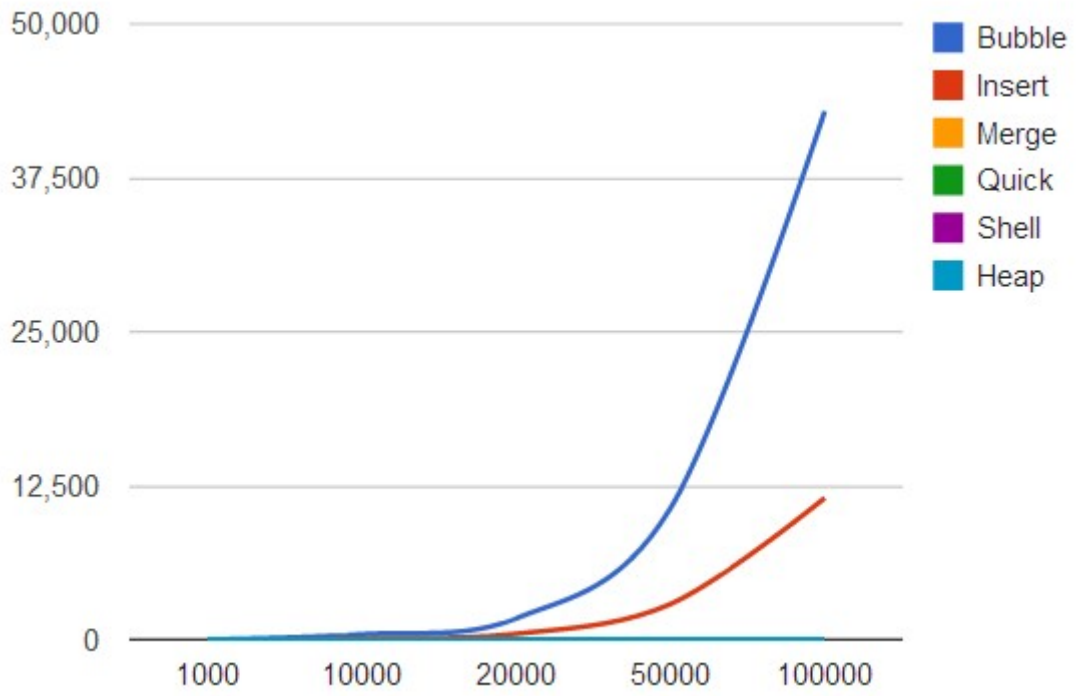
```
            r--;
        if(l<r)
        {
            arr[l]=arr[r];
            l++;
        }

        while(l<r && arr[l]<pivot)
            l++;
        if(l<r)
        {
            arr[r]=arr[l];
            r--;
        }

    }
     arr[l]=pivot;
    return l;
}

void quickSort(int start, int end, int arr[])
{
    int mid;
    if (end > start) {
        mid = partition(start, end ,arr);
        quickSort(start, mid-1, arr);
        quickSort(mid+1, end, arr);
    }
}


//Shell
void shellPass(int arr[],int len,int gap)
{
    int i,j,key;
    //步长不总为1的插入排序
    for ( i = gap; i < len; i++ )
        {
          j = i - gap;
          key = arr[i];
        while (( j >= 0 ) && ( arr[j] > key ))
        {
            arr[j + gap] = arr[j];
            j = j - gap;        //在已排序序列中寻找合适位置插入key
        }
          arr[j + gap] = key;
        }
}

void shellSort(int arr[],int len)
{
    int gap=0;
    while(gap<=len)
        gap=gap*3+1;   //确保最终步长为1
```

```
    while(gap>0)
    {
        shellPass(arr,len,gap);
        gap=(gap-1)/3;
    }
}

//Heap
/*父結點*/
int parent(int i)
{
    return (int)floor((i - 1) / 2);
}

/*左子結點*/
int left(int i)
{
    return (2 * i + 1);
}

/*右子結點*/
int right(int i)
{
    return (2 * i + 2);
}

/*單一子結點最大堆積樹調整*/
void Max_Heapify(int A[], int i, int heap_size)
{
    int l = left(i);
    int r = right(i);
    int largest;
    int temp;
    if(l < heap_size && A[l] > A[i])
    {
        largest = l;
    }
    else
    {
        largest = i;
    }
    if(r < heap_size && A[r] > A[largest])
    {
        largest = r;
    }
    if(largest != i)
    {
        temp = A[i];
        A[i] = A[largest];
        A[largest] = temp;
        Max_Heapify(A, largest, heap_size);
    }
}

/*建立最大堆積樹*/
```

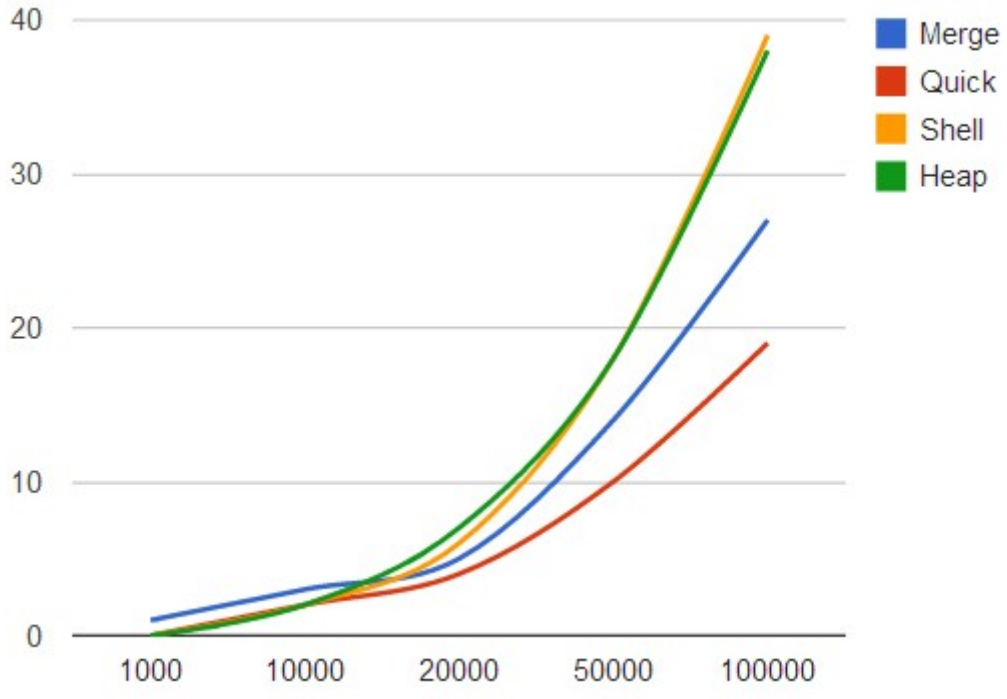
```
void Build_Max_Heap(int A[],int heap_size)
{
  for(int i = (heap_size-2)/2; i >= 0; i--)
  {
      Max_Heapify(A, i, heap_size);
  }
}

/*印出樹狀結構*
void print(int A[], int heap_size)
{
   for(int i = 0; i < heap_size;i++)
  {
      printf("%d ", A[i]);
  }
   printf("\n");
}*/

/*堆積排序程序碼*/
void HeapSort(int A[], int heap_size)
{
   Build_Max_Heap(A, heap_size);
  int temp;
  for(int i = heap_size - 1; i >= 0; i--)
  {
      temp = A[0];
      A[0] = A[i];
      A[i] = temp;
      Max_Heapify(A, 0, i);
  }
  //print(A, heap_size);
}
```

结果分析

所有算法一起比较，简单算法和nlogn的算法的差别非常大：



单独比较nlogn算法：



可见快速排序表现最好

Printed on: 2024/05/19 20:12