**算法分析与设计第二次实验**

1. 实验目的

实现插入排序、自顶向下归并排序、自底向上归并排序、随机快速排序以及Dijkstra 3-路划分快速排序，并对它们的性能进行比较。

二、实验内容

实现插入排序IS，自顶向下归并排序TDM， 自底向上归并排序BUM，随机快速排序RQ，Dijkstra 3-路划分快速排序QD3P。针对不同输入规模数据进行实验，对比上述排序算法的时间及空间占用性能。要求对于每次输入运行 10 次，记录每次时间/空间占用，取平均值。

1. 代码实现

在下面的代码中，我们将各种排序算法封装进了一个类中：

public class Sort {

    public void IS(int[] a){

*//insertion sort；stable*

        int n=a.length;

        for(int i=0;i<n;i++){

            for(int j=i;j>0&&a[j]<a[j-1];j--){

                int t=a[j];

                a[j]=a[j-1];

                a[j-1]=t;

            }

        }

    };

    private void merge(int[] aux,int a[],int lo,int mid,int hi){

        int i=lo,j=mid+1;

        for(int k=lo;k<=hi;k++)

            aux[k]=a[k];

        for(int k=lo;k<=hi;k++)

            if(i>mid) a[k]=aux[j++];

            else if(j>hi) a[k]=aux[i++];

            else if(aux[j]<aux[i]) a[k]=aux[j++];

            else a[k]=aux[i++];

    };

    private void msort(int[] aux,int[] a,int lo,int hi){

        if(hi<=lo) return;

        int mid=(lo+hi)/2;

*//System.out.print(lo+ " "+mid+" "+hi+" \n");*

        msort(aux,a,lo,mid);

        msort(aux,a,mid+1,hi);

        merge(aux,a,lo,mid,hi);

    };

    public void TDM(int[] a){

*//top-down mergesort；stable*

        int aux[]=new int[a.length];

        msort(aux,a,0,a.length-1);

    };

    public void BUM(int[] a){

*//bottom-up mergesort；stable*

        int n=a.length;

        int aux[]=new int[a.length];

        for(int sz=1;sz<n;sz=sz+sz)

            for(int lo=0;lo<n-sz;lo+=sz+sz)

                merge(aux,a,lo,lo+sz-1,Math.min(lo+sz+sz-1,n-1));

    };

    private int partition(int[] a,int lo,int hi){

        int i=lo,j=hi+1,v=a[lo];

        while(true){

            while(a[++i]<v) if(i==hi) break;

            while(v<a[--j]);

            if(i>=j) break;

            int t=a[i];

            a[i]=a[j];

            a[j]=t;

        }

        int t=a[lo];

        a[lo]=a[j];

        a[j]=t;

        return j;

    };

    private void qsort(int[] a,int lo,int hi){

        if(hi<=lo) return;

        int j=partition(a,lo,hi);

        qsort(a,lo,j-1);

        qsort(a,j+1,hi);

    };

    public void RQ(int[] a){

*//random quicksort；not stable*

        qsort(a,0,a.length-1);

    };

    private void q3sort(int[] a,int lo,int hi){

        if(hi<=lo) return;

        int lt=lo,i=lo+1,gt=hi;

        int v=a[lo];

        while(i<=gt){

            if(a[i]<v){

                int t=a[i];

                a[i]=a[lt];

                a[lt]=t;

                lt++;

                i++;

            }else if(a[i]>v){

                int t=a[i];

                a[i]=a[gt];

                a[gt]=t;

                gt--;

            }else{

                i++;

            }

        }

        q3sort(a,lo,lt-1);

        q3sort(a,gt+1,hi);

    };

    public void QD3P(int[] a){

*//Quicksort with Dijkstra 3-way Partition；not stable*

        q3sort(a,0,a.length-1);

    };

    public void print(double[][] b){

        for(int i=0;i<5;i++){

            switch (i){

                case 0:System.out.print("IS   "); break;

                case 1:System.out.print("TDM  "); break;

                case 2:System.out.print("BUM  "); break;

                case 3:System.out.print("RQ   "); break;

                case 4:System.out.print("QD3P "); break;

            }

            for(int j=0;j<11;j++){

                System.out.printf("%.3f",b[i][j]);

                System.out.print("  ");

            }

            System.out.println("");

        }

    };

    public void print(int[] a){

            for(int j=0;j<a.length;j++){

                System.out.printf(a[j]+" ");

            }

            System.out.println("");

    };

    public void copy(int[] s,int[] d){

        for(int j=0;j<s.length;j++){

            d[j]=s[j];

        }

    };

}

上边是算法的实现，下面是对所实现的所有算法的性能测试代码;

import java.util.Scanner;

public class SortTest {

    public static void main(String[] args) {

        double[][] time = new double[5][11];

        double[][] space = new double[5][11];

        Sort s = new Sort();

        int set = new Scanner(System.in).nextInt();

*//重复十次*

        for (int i = 0; i < 10; i++) {

            int[] A = new int[set];

            for (int j = 0; j < set; j++) {

                A[j] = (int) (Math.random() \* (set));

            int[] a = new int[set];

            long startTime = 0;

            long startMem = 0;

            long endMem = 0;

            Runtime r;

            s.copy(A, a);

            startTime = System.currentTimeMillis();

            r = Runtime.getRuntime();

            r.gc();

            startMem = r.totalMemory() - r.freeMemory();

            s.IS(a);

            endMem = r.totalMemory() - r.freeMemory();

            space[0][i] = (endMem - startMem) / 1024;

            time[0][i] = System.currentTimeMillis() - startTime;

            s.print(a);

            s.copy(A, a);

            startTime = System.currentTimeMillis();

            r = Runtime.getRuntime();

            r.gc();

            startMem = r.totalMemory() - r.freeMemory();

            s.TDM(a);

            endMem = r.totalMemory() - r.freeMemory();

            space[1][i] = (endMem - startMem) / 1024;

            time[1][i] = System.currentTimeMillis() - startTime;

            s.copy(A, a);

            startTime = System.currentTimeMillis();

            r = Runtime.getRuntime();

            r.gc();

            startMem = r.totalMemory() - r.freeMemory();

            s.BUM(a);

            endMem = r.totalMemory() - r.freeMemory();

            space[2][i] = (endMem - startMem) / 1024;

            time[2][i] = System.currentTimeMillis() - startTime;

            s.copy(A, a);

            startTime = System.currentTimeMillis();

            r = Runtime.getRuntime();

            r.gc();

            startMem = r.totalMemory() - r.freeMemory();

            s.RQ(a);

            endMem = r.totalMemory() - r.freeMemory();

            space[3][i] = (endMem - startMem) / 1024;

            time[3][i] = System.currentTimeMillis() - startTime;

            s.copy(A, a);

            r = Runtime.getRuntime();

            r.gc();

            startTime = System.currentTimeMillis();

            s.QD3P(a);

            space[4][i] = (endMem - startMem) / 1024;

            time[4][i] = System.currentTimeMillis() - startTime;

        }

        for (int i = 0; i < 5; i++) {

            for (int j = 0; j < 10; j++) {

                time[i][10] += time[i][j];

                space[i][10] += space[i][j];

            }

            time[i][10] = time[i][10] / 10.0;

            space[i][10] = space[i][10] / 10.0;

        }

        System.out.println("\n时间性能对比");

        System.out.println("Run1\tRun2\tRun3\tRun4\tRun5\tRun6\tRun7\tRun8\tRun9\tRun10\tAvg\t");

        s.print(time);

        System.out.println("\n空间性能对比(KB)");

        System.out.println("Run1\tRun2\tRun3\tRun4\tRun5\tRun6\tRun7\tRun8\tRun9\tRun10\tAvg\t");

        s.print(space);

    };

}

}

需要注意的是，为了突出表现3路快排的相较于传统快排的优越性，我设计生成了不少重复的数据，这样可以保证结果的相对公允。

由于代码较为冗长，我们不再去讨论具体的细节，下面我们将进行性能的分析：

根据输出结果，我们得出了如下时间性能对比的表格：（单位ms）

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run1 | Run2 | Run3 | Run4 | Run5 | Run6 | Run7 | Run8 | Run9 | Run10 | Avg |
| IS | 63.5 | 64.7 | 64.1 | 63.3 | 65.1 | 64.1 | 64.9 | 64.9 | 63.7 | 64.1 | 64.2 |
| TDM | 8.0 | 9.2 | 8.6 | 7.8 | 9.6 | 8.6 | 9.2 | 9.4 | 8.2 | 8.6 | 8.7 |
| BUM | 9.6 | 10.8 | 10.2 | 10.1 | 9.4 | 11.2 | 10.2 | 11.0 | 11.0 | 9.8 | 10.3 |
| RQ | 7 | 8.2 | 7.6 | 6.8 | 8.6 | 7.6 | 8.4 | 8.4 | 7.2 | 7.6 | 7.7 |
| QD3P | 5.4 | 6.6 | 6.0 | 5.2 | 7.0 | 6.0 | 6.8 | 6.8 | 5.6 | 6.0 | 6.1 |

以及如下的空间性能对比的表格：（单位KB）

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run1 | Run2 | Run3 | Run4 | Run5 | Run6 | Run7 | Run8 | Run9 | Run10 | Avg |
| IS | 34.2 | 34.0 | 33.8 | 36.6 | 32.2 | 33.4 | 34.5 | 34.4 | 33.4 | 33.8 | 34.0 |
| TDM | 344.4 | 344.6 | 344.0 | 343.2 | 345.0 | 344.0 | 344.8 | 344.8 | 343.6 | 344.0 | 344.2 |
| BUM | 351.4 | 352.6 | 352.0 | 351.2 | 353.0 | 352.0 | 352.8 | 352.8 | 351.6 | 352.0 | 352.1 |
| RQ | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |  |
| QD3P | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |  |

四、问题回答

1. Which sort worked best on data in constant or increasing order (i.e., already sorted data)? Why do you think this sort worked best?

答：插入排序。插入排序对于基本有序的部分只需要少量的比较和移动，在这种情况下，其时间复杂度甚至可以接近O(N)，因此是当前情况下最为高效的排序算法。

2. Did the same sort do well on the case of mostly sorted data? Why or why not?

答：不是这样的。对于插入排序来说，原本的序列越是有序，算法的性能就会越好；而对于快速排序来说，一旦一个序列越是有序，快速排序算法的性能就会越来越差，如果一个序列是完全有序的，那么快速排序算法的性能将跌至O(N²)。

3. In general, did the ordering of the incoming data affect the performance of the sorting algorithms? Please answer this question by referencing specific data from your table to support your answer.

答：为了解答该问题，我写了一个C++的算法程序，使用到了快速排序，分别统计了对有序数组和无序数组的排序时间，程序如下：

#include <vector>

#include <iostream>

#include <time.h>

#include <algorithm>

using namespace std;

*// 分治函数*

int partition(vector<int> &nums, int left, int right)

{

    int pivot = nums[left];

    int i = left, j = right;

    while (i < j)

    {

        while (i < j && nums[j] >= pivot)

        {

            j--;

        }

        nums[i] = nums[j];

        while (i < j && nums[i] <= pivot)

        {

            i++;

        }

        nums[j] = nums[i];

    }

    nums[i] = pivot;

    return i;

}

*// 快速排序函数*

void quickSort(vector<int> &nums, int left, int right)

{

    if (left >= right)

    {

        return;

    }

    int pivotIndex = partition(nums, left, right);

    quickSort(nums, left, pivotIndex - 1);

    quickSort(nums, pivotIndex + 1, right);

}

*// 主函数*

int main()

{

    clock\_t start, end;

    vector<int> nums(10000);

    for (int i = 0; i < 10000; i++)

    {

        nums[i] = 9999 - i;

    }

    start = clock();

    quickSort(nums, 0, nums.size() - 1);

    end = clock();

    cout << "Time consuming:" << (end - start) \* 1000 / CLOCKS\_PER\_SEC <<" ms"<< endl;

    random\_shuffle(nums.begin(), nums.end());

    start = clock();

    quickSort(nums, 0, nums.size() - 1);

    end = clock();

    cout << "Time consuming:" << (end - start) \* 1000 / CLOCKS\_PER\_SEC <<" ms"<< endl;

    return 0;

}

运行结果如下：



上面的输出为完全有序的数组排序所需时间，下面则为打乱的数组排序所需时间，不难看出，二者有着巨大的差异。

4. Which sort did best on the shorter (i.e., n = 1,000) data sets? Did the same one do better on the longer (i.e., n = 100,000) data sets? Why or why not? Please use specific data from your table to support your answer.

答：在数据较小时，这几种排序算法的时间差异并不大，但是随着数据量的增加，插入排序的性能最差，其他排序算法有些许差异但是并不明显。

5. In general, which sort did better? Give a hypothesis as to why the difference in performance exists.

答：一般而言，快速排序可以满足我们的大部分需求，因为它便于设计，且运行性能较好，三路快排是基于快速排序的改良，它对于大量重复数据有着更加高的排序效率，但是它是不稳定的排序方法并且它的性能最差可以到达O(N²)。与之相比，归并排序虽然在平均性能上较弱于快速排序，但是最差性能却比快速排序要好，因此使用归并排序可以规避掉一些极端情况下的低效率事件出现，总之，没有最好的算法，只有最适合的算法。

6. Are there results in your table that seem to be inconsistent? (e.g., If I get run times for a sort that look like this {1.3, 1.5, 1.6, 7.0, 1.2, 1.6, 1.4, 1.8, 2.0, 1.5] the 7.0 entry is not consistent with the rest). Why do you think this happened？

答：我认为这可能与计算机内部的某些硬件设计有关，或者这与操作系统的设计有着很大的关系，并不是我们的算法出了差错。