CS590 Homework 1

Zhaocong Wang

To make testing and computing of average running time for each combination of algorithm, The following solution is adopted:

In order to avoid repeatedly and manually run the program to get actual running time for each combination and then calculate the average ones, a for loop mechanism is used to run an algorithm multiple times and then calculate the average running time automatically and efficiently. The only thing to do is to provide how many times you want an algorithm to run.
For example, I write the following code in the last of main.cpp.

I set the running time as 10, and generate a vector, then put it into the for loop to execute for 10 times and calculate the average time.
(D1,D2 will be changed and the type of vector can be replaced)

```
//int main()
//{
    // int D1 = 2500000;
    // int D2 = 10;
    // int imT = 10;
      //int** pAA = sort::create_random_ivector(D2, D1,true);
      //int** pAA = sort::create_sorted_ivector(D2, D1);
      //int** pAA = sort::create_reverse_sorted_ivector(D2, D1);
    // struct timeval beforeNaive;
    // gettimeofday(&beforeNaive,NULL);
    // for (int i = 0; i < imT; ++i)
  //    {
//
//     sort::insertion_sort(pAA, D2, 0, D1 - 1);
      //    sort::merge_sort(pAA,D1,D2);
      //
    // }
    // struct timeval afterNaive;
    // gettimeofday(&afterNaive,NULL);
    //    double time = (afterNaive.tv_sec*1000 + afterNaive.tv_usec/1000) -
(beforeNaive.tv_sec*1000 + beforeNaive.tv_usec/1000);
    // printf("Naive Isort - Time: %.2f ms. D1: %d. D2: %d.\n", time / imT, D1, D2);
    // printf("Merge Isort - Time: %.2f ms. D1: %d. D2: %d.\n", time / imT, D1, D2);
    //sort::remove_ivector(pAA,D1);
      //int** pIM = sort::create_random_ivector(D2, D1, true);
```

```
      //int** pIM = sort::create_sorted_ivector(D2, D1);
    // int** pIM = sort::create_reverse_sorted_ivector(D2, D1);
    // struct timeval beforeIM;
    // gettimeofday(&beforeIM, NULL);
    // for (int i = 0; i < imT; ++i)
  //    {
        //      sort::insertion_sort_im(pIM, D2, 0, D1 - 1);
    // }
    // struct timeval afterIM;
    // gettimeofday(&afterIM, NULL);
    // double timeIM = (afterIM.tv_sec * 1000 + afterIM.tv_usec / 1000) - (beforeIM.tv_sec *
1000 + beforeIM.tv_usec / 1000);
    // printf("im Isort - Time: %.2f ms. D1: %d. D2: %d.\n", timeIM / imT, D1, D2);
//}
```

The runtime:

Merge Sort:

Random:

Merge Isort - Time: 6.10 ms. D1: 10000. D2: 10.
Merge Isort - Time: 7.00 ms. D1: 10000. D2: 25.
Merge Isort - Time: 10.00 ms. D1: 10000. D2: 50.
Merge Isort - Time: 32.20 ms. D1: 25000. D2: 10
Merge Isort - Time: 33.10 ms. D1: 25000. D2: 25.
Merge Isort - Time: 37.60 ms. D1: 25000. D2: 50.
Merge Isort - Time: 123.70 ms. D1: 50000. D2: 10
Merge Isort - Time: 133.70 ms. D1: 50000. D2: 25
Merge Isort - Time: 145.90 ms. D1: 50000. D2: 50
Merge Isort - Time: 483.90 ms. D1: 100000. D2: 10.
Merge Isort - Time: 511.40 ms. D1: 100000. D2: 25.
Merge Isort - Time: 496.50 ms. D1: 100000. D2: 50
Merge Isort - Time: 3479.20 ms. D1: 250000. D2: 10.
Merge Isort - Time: 3399.30 ms. D1: 250000. D2: 25.
Merge Isort - Time: 3479.20 ms. D1: 250000. D2: 50.
Merge Isort - Time: 13526.90 ms. D1: 500000. D2: 10
Merge Isort - Time: 13586.90 ms. D1: 500000. D2: 25
Merge Isort - Time: 13795.10 ms. D1: 500000. D2: 50.

Sorted

Merge Isort - Time: 2.30 ms. D1: 10000. D2: 10.

Merge Isort - Time: 4.70 ms. D1: 10000. D2: 25.
Merge Isort - Time: 6.50 ms. D1: 10000. D2: 50
Merge Isort - Time: 5.00 ms. D1: 25000. D2: 10.
Merge Isort - Time: 8.50 ms. D1: 25000. D2: 25.
Merge Isort - Time: 14.50 ms. D1: 25000. D2: 50.
Merge Isort - Time: 8.00 ms. D1: 50000. D2: 10.
Merge Isort - Time: 15.30 ms. D1: 50000. D2: 25.
Merge Isort - Time: 25.50 ms. D1: 50000. D2: 50
Merge Isort - Time: 116.00 ms. D1: 250000. D2: 50.
Merge Isort - Time: 221.10 ms. D1: 500000. D2: 50.
Merge Isort - Time: 1626.90 ms. D1: 2500000. D2: 50.

Reverse sort
Merge Isort - Time: 11.70 ms. D1: 10000. D2: 10.
Merge Isort - Time: 13.90 ms. D1: 10000. D2: 25.
Merge Isort - Time: 16.50 ms. D1: 10000. D2: 50.
Merge Isort - Time: 65.20 ms. D1: 25000. D2: 10.
Merge Isort - Time: 68.10 ms. D1: 25000. D2: 25
Merge Isort - Time: 76.90 ms. D1: 25000. D2: 50.
Merge Isort - Time: 268.20 ms. D1: 50000. D2: 10.
Merge Isort - Time: 298.10 ms. D1: 50000. D2: 25
Merge Isort - Time: 288.20 ms. D1: 50000. D2: 50
Merge Isort - Time: 1078.60 ms. D1: 100000. D2: 10.
Merge Isort - Time: 1057.70 ms. D1: 100000. D2: 25
Merge Isort - Time: 1111.60 ms. D1: 100000. D2: 50
Merge Isort - Time: 6524.80 ms. D1: 250000. D2: 10
Merge Isort - Time: 6937.80 ms. D1: 250000. D2: 25.
Merge Isort - Time: 6971.50 ms. D1: 250000. D2: 50
Merge Isort - Time: 27915.80 ms. D1: 500000. D2: 10.

From the performance of random, sorted and reverse sorted input, it is clear to see that
sorted< random< reverse

For the merge sort algorithm, compared to the second-dimension length of the two-dimensional array, when the length of the first-dimension of the two-dimensional array increases, the required average running time will increase more significantly. The value of D2 has a little effect when D1 doesn't change. For example:
Merge Isort - Time: 6.10 ms. D1: 10000. D2: 10.
Merge Isort - Time: 7.00 ms. D1: 10000. D2: 25.
Merge Isort - Time: 32.20 ms. D1: 25000. D2: 10
Merge Isort - Time: 33.10 ms. D1: 25000. D2: 25.

It shows that when D1=10000, D2=25 and D1=25000, D2=10, they have the same amount of

elements, but their times have a difference.

As a conclusion: The effect of the length of the first dimension of the two-dimensional array on the average time will be much greater than the length of the second dimension. Therefore, when the sorting algorithm is applied, when the total number of elements is constant, the length of the first dimension should be reduced as much as possible to increase the length of the second dimension. It will be more efficiency.

Insertion Sort with naïve and im:

Random

Naive Isort - Time: 184.20 ms. D1: 10000. D2: 10.
Naive Isort - Time: 395.40 ms. D1: 10000. D2: 25
Naive Isort - Time: 849.00 ms. D1: 10000. D2: 50.
Naive Isort - Time: 1064.00 ms. D1: 25000. D2: 10.
Naive Isort - Time: 2873.50 ms. D1: 25000. D2: 25.
Naive Isort - Time: 5661.70 ms. D1: 25000. D2: 50.
Naive Isort - Time: 4930.30 ms. D1: 50000. D2: 10.
Naive Isort - Time: 12962.80 ms. D1: 50000. D2: 25.

im Isort - Time: 12.20 ms. D1: 10000. D2: 10..
im Isort - Time: 11.80 ms. D1: 10000. D2: 25.
im Isort - Time: 13.70 ms. D1: 10000. D2: 50.
im Isort - Time: 63.30 ms. D1: 25000. D2: 10.
im Isort - Time: 67.70 ms. D1: 25000. D2: 25.
im Isort - Time: 66.90 ms. D1: 25000. D2: 50.
im Isort - Time: 247.10 ms. D1: 50000. D2: 10.
im Isort - Time: 267.20 ms. D1: 50000. D2: 25.
im Isort - Time: 1078.90 ms. D1: 100000. D2: 10.
im Isort - Time: 1100.10 ms. D1: 100000. D2: 25..
im Isort - Time: 1037.50 ms. D1: 100000. D2: 50.
im Isort - Time: 6662.70 ms. D1: 250000. D2: 10.
im Isort - Time: 6835.40 ms. D1: 250000. D2: 25
im Isort - Time: 6962.00 ms. D1: 250000. D2: 50
im Isort - Time: 28091.70 ms. D1: 500000. D2: 10.

Sorted:

Naive Isort - Time: 4.20 ms. D1: 10000. D2: 10.
Naive Isort - Time: 5.60 ms. D1: 10000. D2: 25.

Naive Isort - Time: 8.50 ms. D1: 10000. D2: 50.
Naive Isort - Time: 6.30 ms. D1: 25000. D2: 10.
Naive Isort - Time: 13.80 ms. D1: 25000. D2: 25.
Naive Isort - Time: 20.90 ms. D1: 25000. D2: 50.
Naive Isort - Time: 13.40 ms. D1: 50000. D2: 10
Naive Isort - Time: 39.20 ms. D1: 50000. D2: 50.
Naive Isort - Time: 20.60 ms. D1: 100000. D2: 10.
Naive Isort - Time: 42.60 ms. D1: 100000. D2: 25.
Naive Isort - Time: 82.70 ms. D1: 100000. D2: 50.
Naive Isort - Time: 52.20 ms. D1: 250000. D2: 10.
Naive Isort - Time: 241.60 ms. D1: 250000. D2: 50.
Naive Isort - Time: 516.30 ms. D1: 2500000. D2: 10.
Naive Isort - Time: 1185.00 ms. D1: 2500000. D2: 25.

im Isort - Time: 0.40 ms. D1: 10000. D2: 10.
im Isort - Time: 1.40 ms. D1: 10000. D2: 25.
im Isort - Time: 2.00 ms. D1: 10000. D2: 50.
im Isort - Time: 1.20 ms. D1: 25000. D2: 10.
im Isort - Time: 3.00 ms. D1: 25000. D2: 25.
im Isort - Time: 5.00 ms. D1: 25000. D2: 50.
im Isort - Time: 3.20 ms. D1: 50000. D2: 10.
im Isort - Time: 8.10 ms. D1: 50000. D2: 50.
im Isort - Time: 4.30 ms. D1: 100000. D2: 10.
im Isort - Time: 7.70 ms. D1: 100000. D2: 25.
im Isort - Time: 17.40 ms. D1: 100000. D2: 50.
im Isort - Time: 11.60 ms. D1: 250000. D2: 10.
im Isort - Time: 46.40 ms. D1: 250000. D2: 50.
im Isort - Time: 101.90 ms. D1: 2500000. D2: 10.
im Isort - Time: 184.30 ms. D1: 2500000. D2: 25.

Reserve sorted
Naive Isort - Time: 288.90 ms. D1: 10000. D2: 10.
Naive Isort - Time: 719.20 ms. D1: 10000. D2: 25.
Naive Isort - Time: 1376.40 ms. D1: 10000. D2: 50.
Naive Isort - Time: 1748.90 ms. D1: 25000. D2: 10.
Naive Isort - Time: 4205.10 ms. D1: 25000. D2: 25.
Naive Isort - Time: 8497.90 ms. D1: 25000. D2: 50.
Naive Isort - Time: 6784.80 ms. D1: 50000. D2: 10.
Naive Isort - Time: 16923.00 ms. D1: 50000. D2: 25.

im Isort - Time: 20.00 ms. D1: 10000. D2: 10.
im Isort - Time: 21.80 ms. D1: 10000. D2: 25.

im Isort - Time: 21.50 ms. D1: 10000. D2: 50.
im Isort - Time: 134.00 ms. D1: 25000. D2: 10.
im Isort - Time: 131.80 ms. D1: 25000. D2: 25.
im Isort - Time: 129.80 ms. D1: 25000. D2: 50.
im Isort - Time: 505.20 ms. D1: 50000. D2: 10.
im Isort - Time: 501.60 ms. D1: 50000. D2: 25.
im Isort - Time: 509.30 ms. D1: 50000. D2: 50.
im Isort - Time: 2098.90 ms. D1: 100000. D2: 10.
im Isort - Time: 2328.30 ms. D1: 100000. D2: 25
im Isort - Time: 2057.80 ms. D1: 100000. D2: 50.

From the performance of random, sorted and reverse sorted input, it is clear to see that sorted< random< reverse in both naïve and im method.

For naïve sort, the first dimension and the second dimension all affect the running time a lot. And it takes a long time to get result.

The following results can be obtained for im insertion sorting and naïve insertion sorting in which the length is calculated in advance.
Naive Isort - Time: 184.20 ms. D1: 10000. D2: 10.
im Isort - Time: 12.20 ms. D1: 10000. D2: 10..

As a conclusion: the pre-calculated length greatly improves the average running time of the algorithm for large-length arrays, and the efficiency is improved by nearly 10 times.

The analysis of error:

I set the function of vector generation out of the loop, so the array will be sorted after the first time in loop. As a result, it is hard to make a quantitative analysis.
If I set the function of vector generation in the loop, it will take more time on generation and removing.

Improvement:

Values that are reused multiple times should be calculated in advance and saved in memory. This is the typical "space change strategy". Especially for large-scale data that looks like a dictionary, pre-calculation and storage in memory will greatly improve efficiency, usually up to tens of times.