Algorithm Analysis and Design Lab1

实验内容

本次实验要求实现以下几种排序算法：

- 插入排序
- 归并排序
- 快速排序
- 堆排序
- 基数排序
- 桶排序

并对其进行时间消耗的分析

实验过程

- 随机数组生成

  这里我选择使用C语言的 `rand()` 函数进行数组元素的生成。`rand()` 函数生成的数据范围为 `0-32767`

  ```
  void rand_gen(int *array, int n)
  {
      srand((unsigned)time(NULL));
      for (int i = 0; i < n; i++)
      {
          array[i] = rand();
      }
  }
  ```

- 基本有序数组生成

  这里我将每100个元素限制在其范围内，由此生成基本有序数组：

  ```
  void nearly_sorted_gen(int *array, int n)
  {
      srand((unsigned)time(NULL));
      int i;
      int slice = n/100;

      for (i = 0; i < n; i++)
      {
          array[i] = rand()%slice + (i/slice)*100;
      }
  }
  ```

- 插入排序

  插入排序的代码实现较为简单，展示如下：

  ```
  void Insert_Sort(int *array, int n)
  ```

```
{
    int i;
    for (i = 1; i < n; i++)
    {
        int tmp = array[i];
        int j = i;

        while (array[j-1] > tmp && j > 0)
        {
            array[j] = array[j - 1];
            j--;
        }
        array[j] = tmp;
    }
}
```

- 归并排序

  这里我们采用2路归并算法，递归实现，具体函数如下：

```
void merge_sort(int *array, int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        merge_sort(array, l, m);
        merge_sort(array, m + 1, r);

        merge(array, l, r);
    }
}

void merge(int *array, int l, int r)
{
    int i, j, k;
    int m = l + (r - l) / 2;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = array[l + i];
    for (j = 0; j < n2; j++)
        R[j] = array[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
```

```
            array[k] = L[i];
            i++;
        }
        else
        {
            array[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        array[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        array[k] = R[j];
        j++;
        k++;
    }
}
```

- 快速排序

  在我实现的快速排序中，我采取每个数组的第一个元素作为枢轴来对数组进行分隔，分隔数组的函数如下：

```
int partition(int *array, int low, int high)
{
    int pivotkey = array[low];
    int tmp;
    while (low < high)
    {
        while (low < high && array[high] >= pivotkey)
            --high;
        tmp = array[high];
        array[high] = array[low];
        array[low] = tmp;

        while (low < high && array[low] <= pivotkey)
            ++low;
        tmp = array[high];
        array[high] = array[low];
        array[low] = tmp;
    }
    return low;
}
```

  通过调用 `partition` 函数，我递归实现了快速排序，具体函数如下：

```
void quick_sort(int *array, int low, int high)
{
    if (low < high)
    {
        int pivotloc = partition(array, low, high);
        quick_sort(array, low, pivotloc - 1);
        quick_sort(array, pivotloc + 1, high);
    }
}
```

- 堆排序：

  对于堆排序，我们首先定义这样一个将数组元素沉底的函数：

```
void heapify(int *array, int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;
    int tmp;

    if (l < n && array[l] > array[largest])
        largest = l;

    if (r < n && array[r] > array[largest])
        largest = r;

    if (largest != i)
    {
        tmp = array[i];
        array[i] = array[largest];
        array[largest] = tmp;
        heapify(array, n, largest);
    }
}
```

  通过调用以上函数，我们可以将一个数组逐元素调整为堆，取出堆顶元素，将堆底元素替换之，再进行沉底，如此往复，我们就可以将数组调整为有序，具体代码实现如下：

```
void heap_sort(int *array, int n)
{
    int tmp;
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(array, n, i);

    for (int i=n-1; i>=0; i--)
    {
        tmp = array[i];
        array[i] = array[0];
        array[0] = tmp;
        heapify(array, i, 0);
    }
}
```

- 基数排序

  对于基数排序，选择以10为基数，从个位开始，逐位对于数据进行排序，最终得到的数组即为有序的。

  首先定义这样一个函数，用于每个基数内的排序：

```
void countSort(int arr[], int n, int exp)
{
    int *output = (int *)malloc(n * sizeof(int));
    int i, count[10] = {0};

    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

  其思想为，首先统计位数为每个数字的元素个数，从而找到在输出数组中的对应位置，然后逐个输出即可。通过调用以上函数，我们可以逐位实现排序：

```
void radix_sort(int arr[], int n)
{
    int m = getMax(arr, n);
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

- 桶排序

  在桶排序中，我这里采用 n 个桶，桶的被定义为结构体：

```
typedef struct
{
    int *array;
    int length;
}bucket;
```

  其中 `array` 为动态分配的数组，这样便于元素的添加。

  向某一个桶中添加元素的函数为

```
void bucket_insert(bucket *bucket_list, int max, int data, int n)
{
    int i = min_((int)((float)data/(float)max * (float)n), n - 1);
    bucket_list[i].array = (int *)realloc(bucket_list[i].array,
(bucket_list[i].length + 1) * sizeof(int));
    bucket_list[i].array[bucket_list[i].length] = data;
    bucket_list[i].length += 1;
}
```

在按照不同桶将数据归类后，我使用快速排序来进行桶内的排序，并最终将所有桶合并为一个数组输出：

```
void bucket_sort(int *array, int n)
{
    bucket *bucket_list = (bucket *)malloc(n * sizeof(bucket));
    int size = sizeof(bucket_list);
    for (int i = 0; i < n; i++)
    {
        bucket_list[i].array = (int *)malloc(sizeof(int));
        bucket_list[i].length = 0;
    }

    int max = getMax(array, n);

    for (int i = 0; i < n; i++)
        bucket_insert(bucket_list, max, array[i], n);

    for (int i = 0; i < n; i++)
        if (bucket_list[i].length > 0)
            quick_sort(bucket_list[i].array, 0, bucket_list[i].length);

    int index = 0;
    for (int i = 0; i < n; i++)
        if (bucket_list[i].length > 0)
            for (int j = 0; j < bucket_list[i].length; j++)
            {
                array[index] = bucket_list[i].array[j];
                index++;
            }
}
```

运行结果

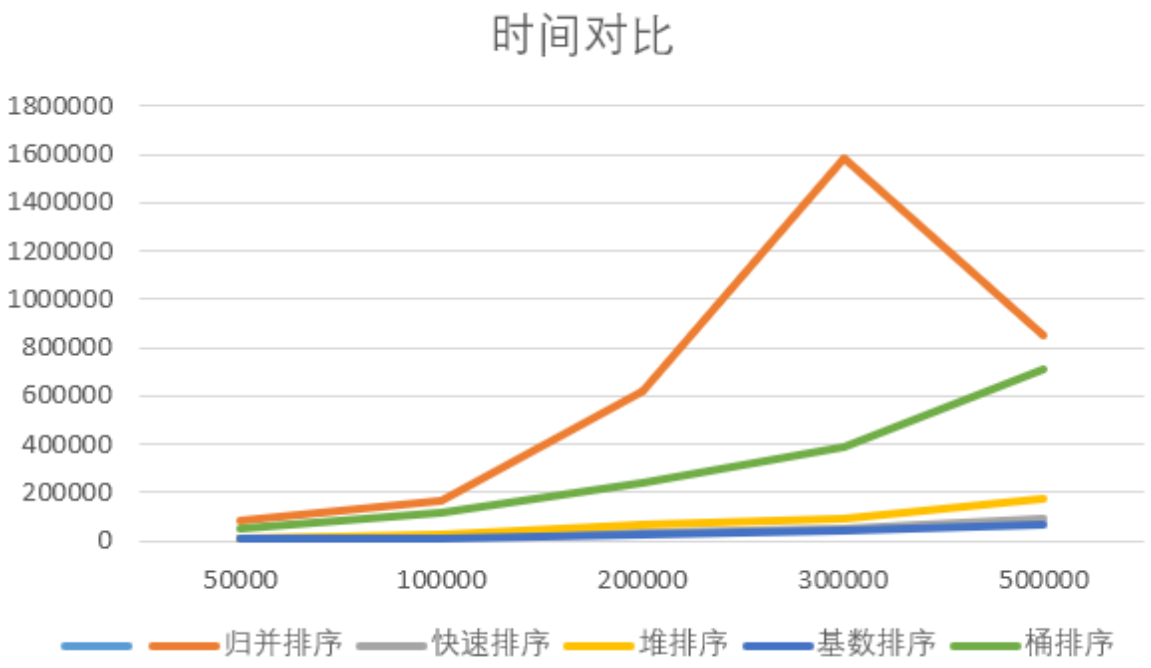
首先是针对不同规模大小数组的排序时间：

```
Array size: 50000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              2051446.900000
Merge_Sort               81537.500000
Quick_Sort               7987.900000
Heap_Sort                12741.000000
Radix_Sort               6320.100000
Bucket_Sort              49150.400000
```

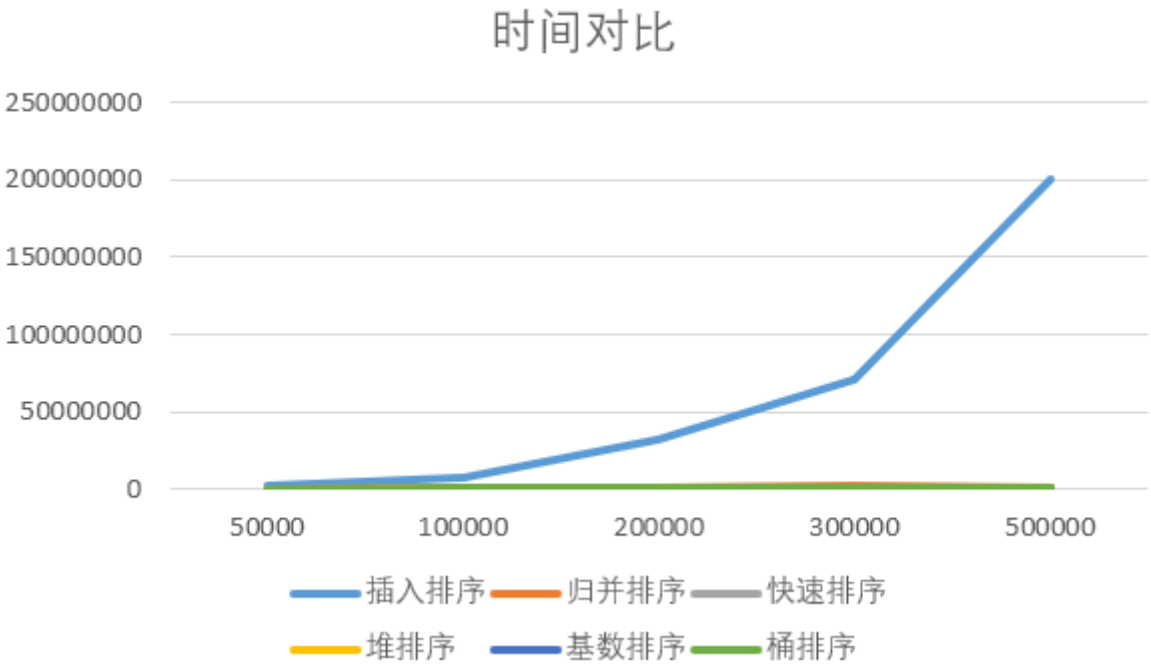
```
Array size: 100000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              7825410.900000
Merge_Sort               169724.500000
Quick_Sort               15825.200000
Heap_Sort                29419.000000
Radix_Sort               13837.800000
Bucket_Sort              119236.100000

Array size: 200000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              31359604.000000
Merge_Sort               618954.400000
Quick_Sort               33773.500000
Heap_Sort                64081.100000
Radix_Sort               26927.500000
Bucket_Sort              237114.500000

Array size: 300000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              70731552.000000
Merge_Sort               1582993.000000
Quick_Sort               54977.700000
Heap_Sort                94827.500000
Radix_Sort               43290.500000
Bucket_Sort              389790.900000

Array size: 500000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              201112715.600000
Merge_Sort               854068.200000
Quick_Sort               89123.400000
Heap_Sort                174557.600000
Radix_Sort               65958.800000
Bucket_Sort              709872.100000
```

绘图如下：

由于插入排序时间消耗过大，先将其排除，对其他几种排序进行比较：



加上插入排序的结果为：

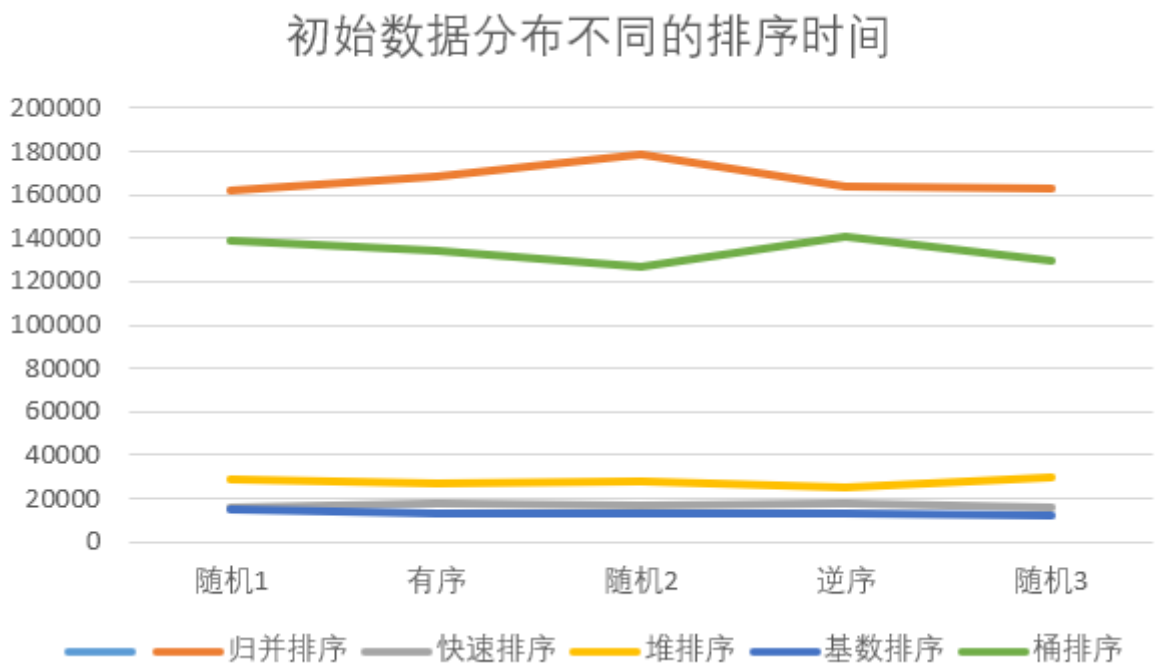


然后是针对不同类型的数据，使用不同算法进行排序，消耗时间如下：

```
Array size: 100000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              8529209.600000
Merge_Sort               162219.400000
Quick_Sort               16039.400000
Heap_Sort                29271.300000
Radix_Sort               15149.400000
Bucket_Sort              138450.000000

Array size: 100000 (nearly sorted)
Sort_Algorithm          time_used(microseconds)
Insert_Sort              552999.700000
```

```
Merge_Sort                168766.500000
Quick_Sort                17379.200000
Heap_Sort                 26865.000000
Radix_Sort                12980.700000
Bucket_Sort               133909.000000

Array size: 100000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort               8313981.500000
Merge_Sort                178218.900000
Quick_Sort                16851.900000
Heap_Sort                 27987.300000
Radix_Sort                12747.400000
Bucket_Sort               126792.200000

Array size: 100000 (nearly reversed sorted)
Sort_Algorithm          time_used(microseconds)
Insert_Sort               15483644.000000
Merge_Sort                164053.600000
Quick_Sort                17893.400000
Heap_Sort                 25042.400000
Radix_Sort                12840.600000
Bucket_Sort               140922.200000

Array size: 100000 (randomly generated)
Sort_Algorithm          time_used(microseconds)
Insert_Sort               7864749.700000
Merge_Sort                163210.200000
Quick_Sort                15793.300000
Heap_Sort                 29673.800000
Radix_Sort                12651.500000
Bucket_Sort               129864.600000
```

排除插入排序的结果为：



加上插入排序的结果为：


初始数据分布不同的排序时间
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
随机1
有序
随机2
逆序
随机3
插入排序
归并排序
快速排序
堆排序
基数排序
桶排序