实验1

1、实验要求

PPT上已给出，略去不表。

2、实验环境

编译环境：Xcode 11.0

机器内存：8G

时钟主频：2.9GHz

3、实验过程

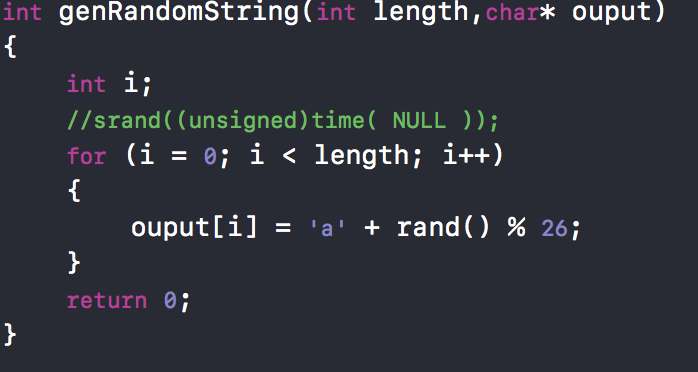
exp1首先实现了创建随机字符串的过程，实验中中各函数的实现基本与课本给出的排序算法相一致，实验中实现了一个greater（char **a, char**\*b）函数，用来比较两个字符串的大小关系。

exp2首先实现了创建随机数字的过程，实验中radix\_sort (int \*source, int n）函数中创建了一个int temp[n][5]数组，用于保存source数组中各个位置的大小，基数排序使用了冒泡排序为各位置从低位至高位排序；其余排序与课本基本类似。

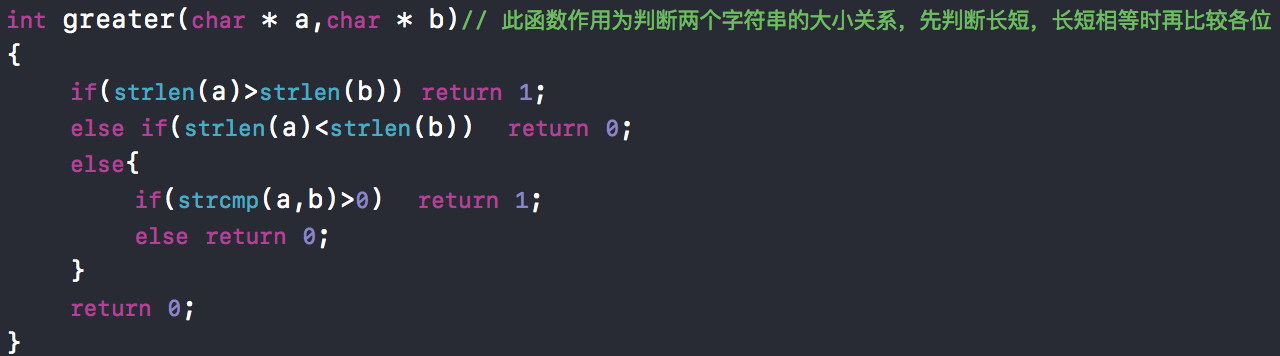
函数运行时间的计算由clock（）函数完成。

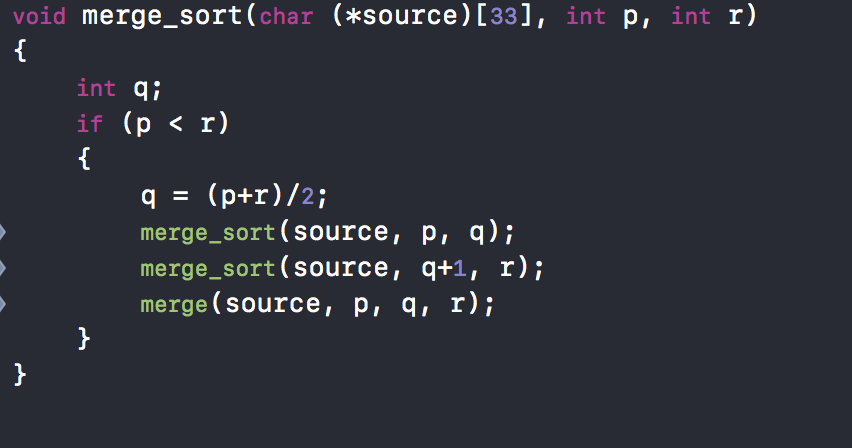
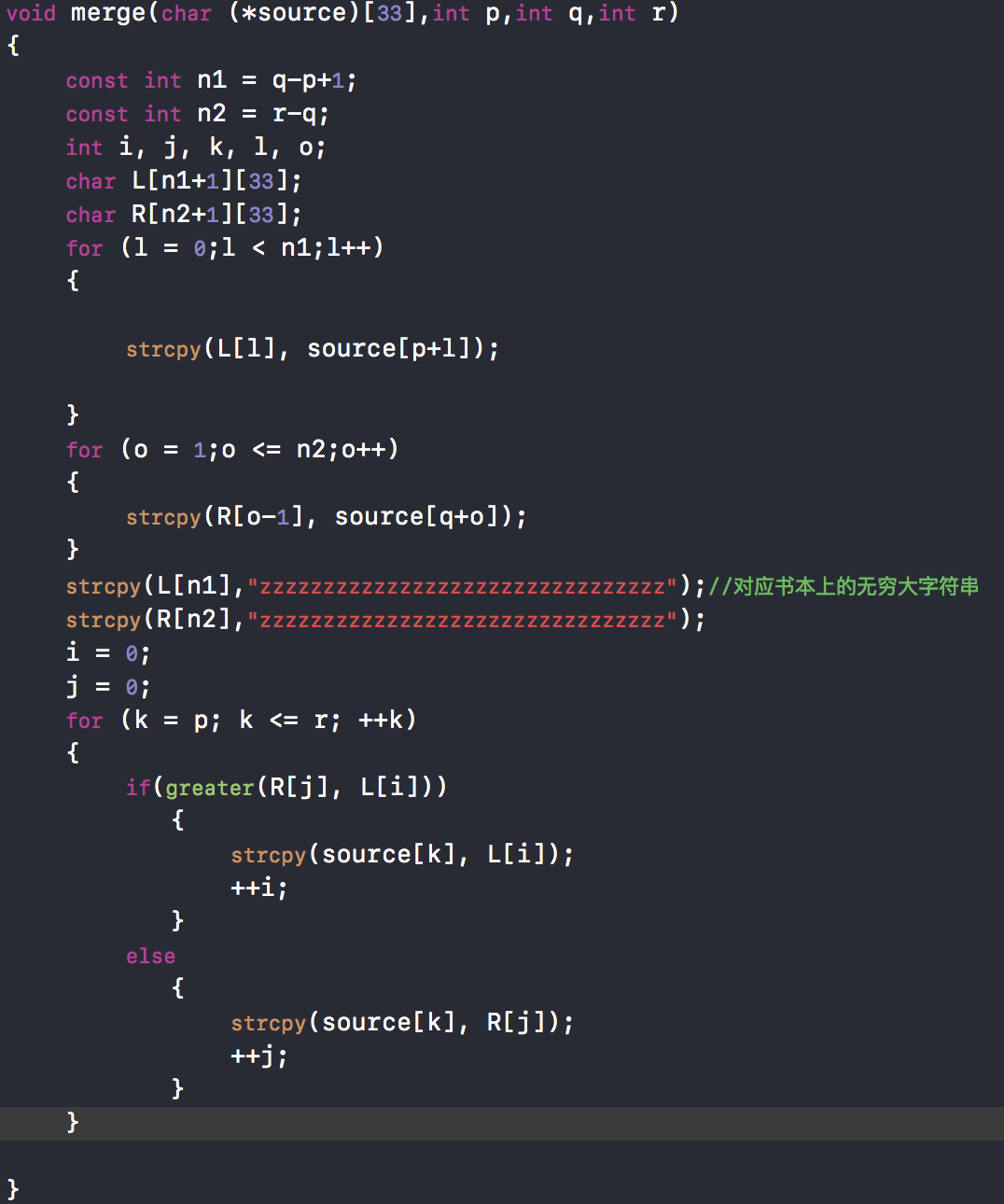
4、实验关键代码截图

4.1 exp1

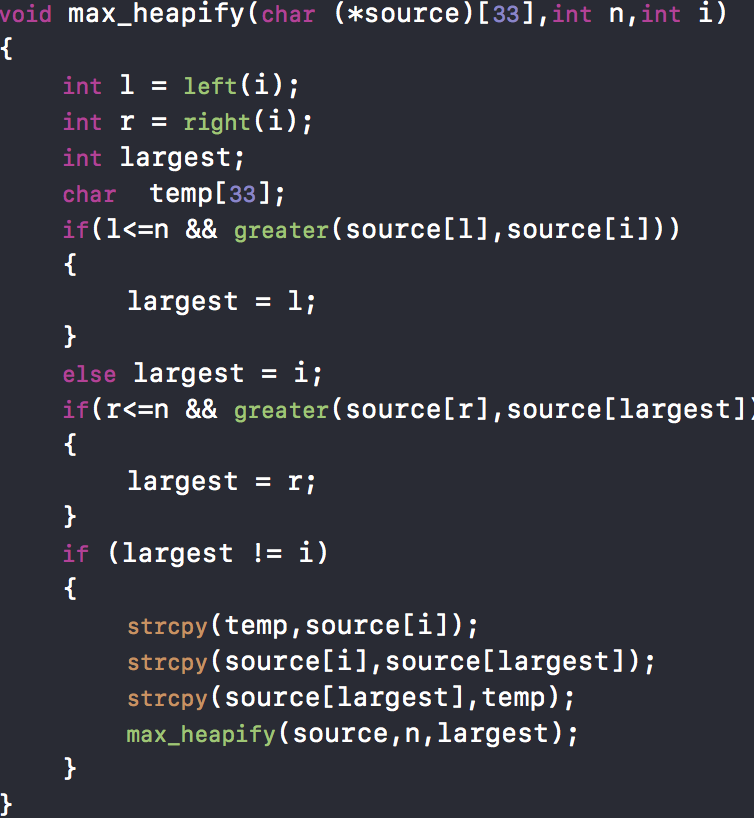
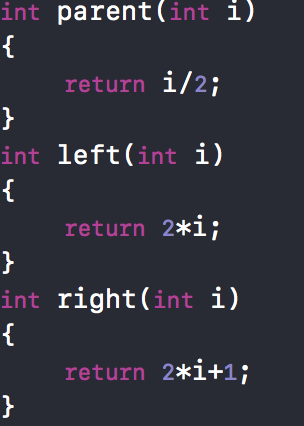


此函数用于生成随机字符串



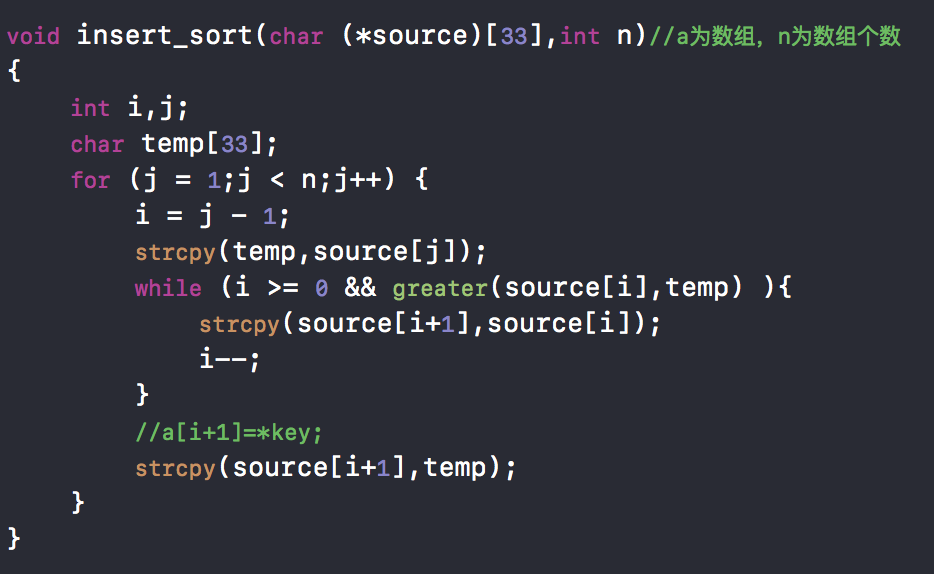


以上为merge\_sort函数，此函数的原理在于递归地将待排序的字符串分成两半进行排序，此排序的时间复杂度为O（nlogn）

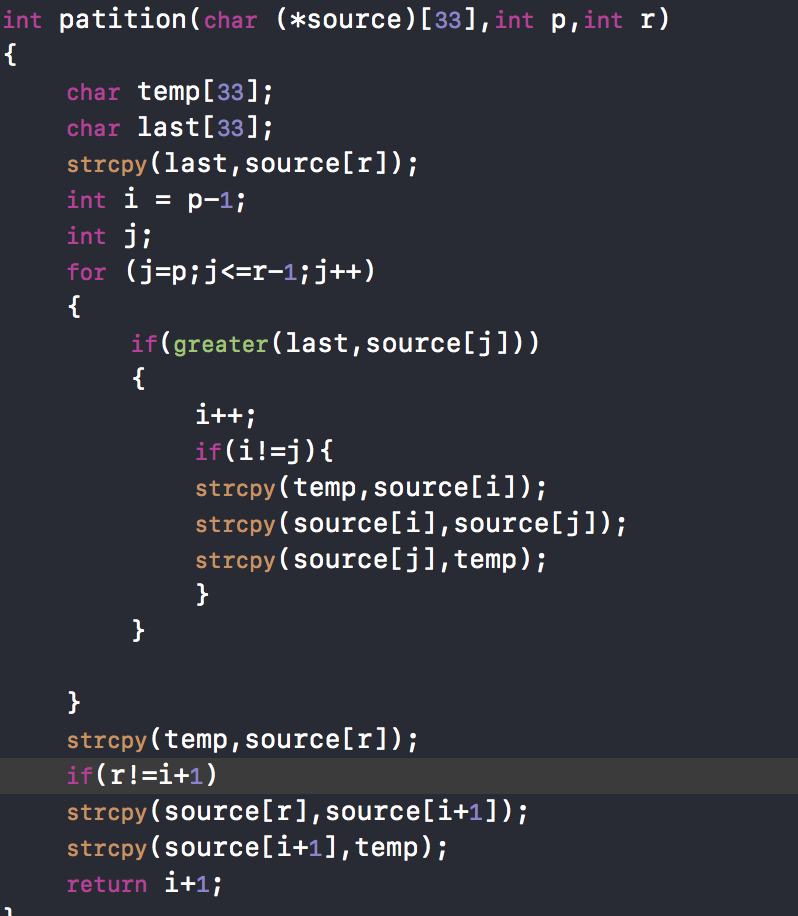
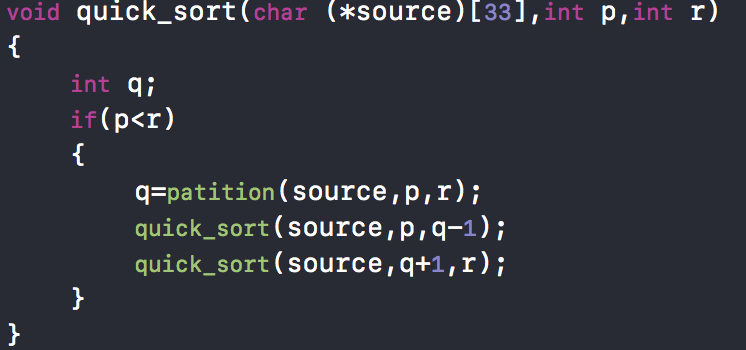




堆排序的原理在于先构造一个大根堆，再依次执行取出根节点又维护大根堆的操作，这样就能实现堆排序，时间复杂度为O（nlogn）

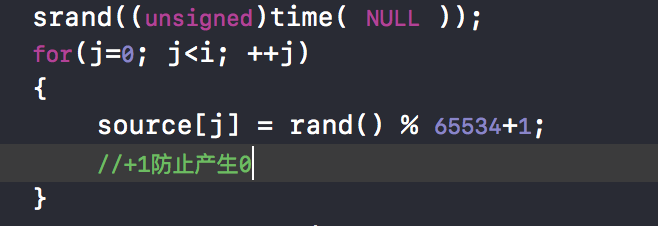


插入排序的过程最为简单，只要从后往前找到比哨兵元素大的元素，将其往后移一个位置，再将哨兵元素插入对应位置即可，时间复杂度为O（n2）。

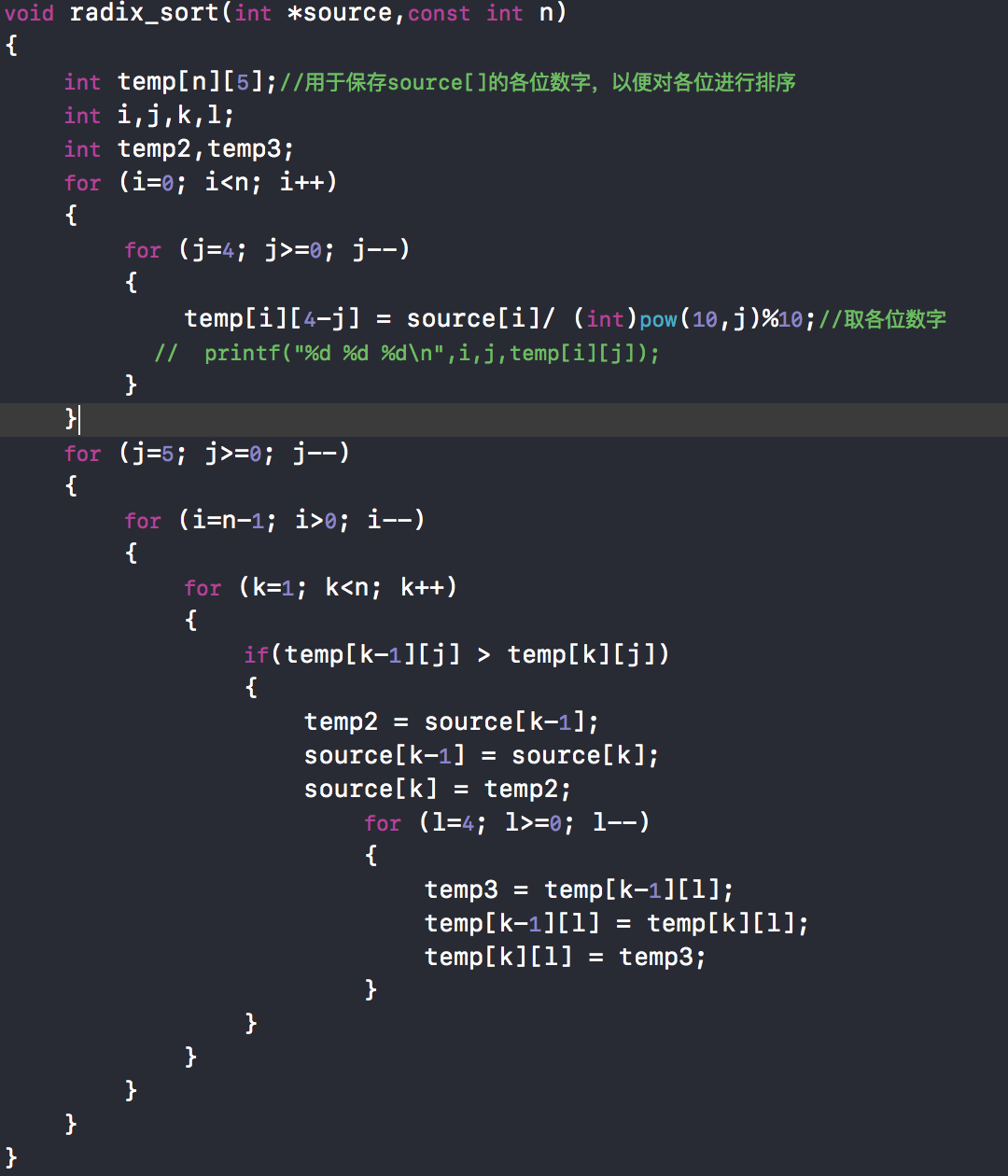
 

快排的原理在于partition函数可以确定一个哨兵，在这个哨兵前面的元素全部小于这个哨兵，而在这个哨兵后面的元素全都大于这个哨兵，确定了哨兵以后只要递归调用quick\_sort就可以完成排序，时间复杂度为O（nlogn）。

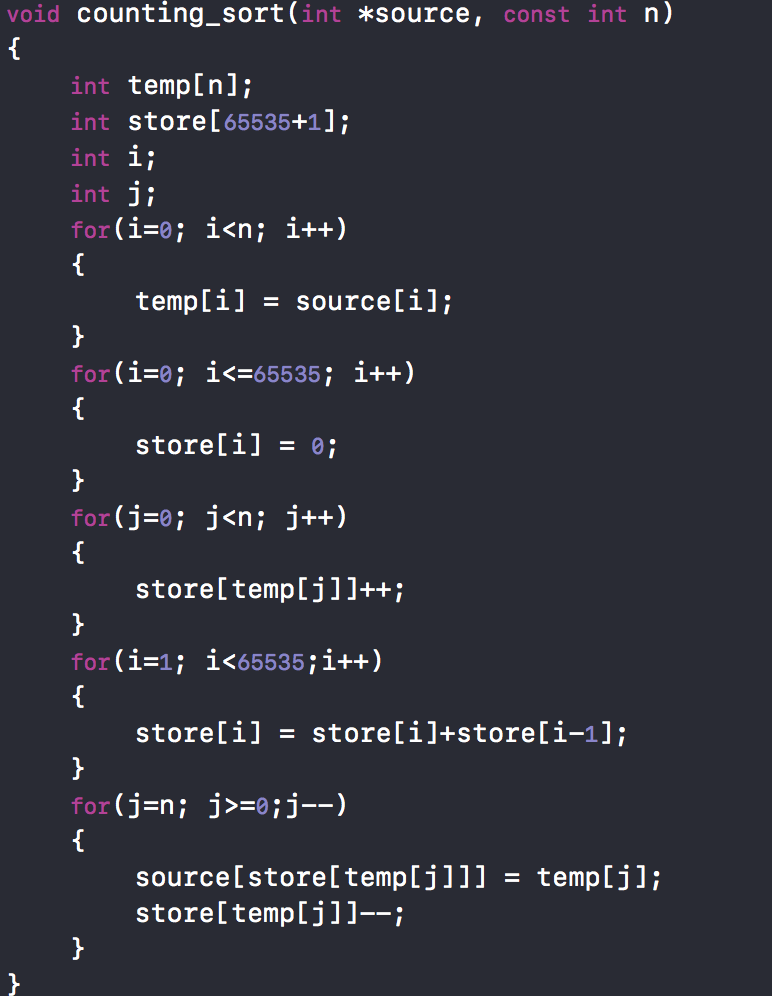
exp2



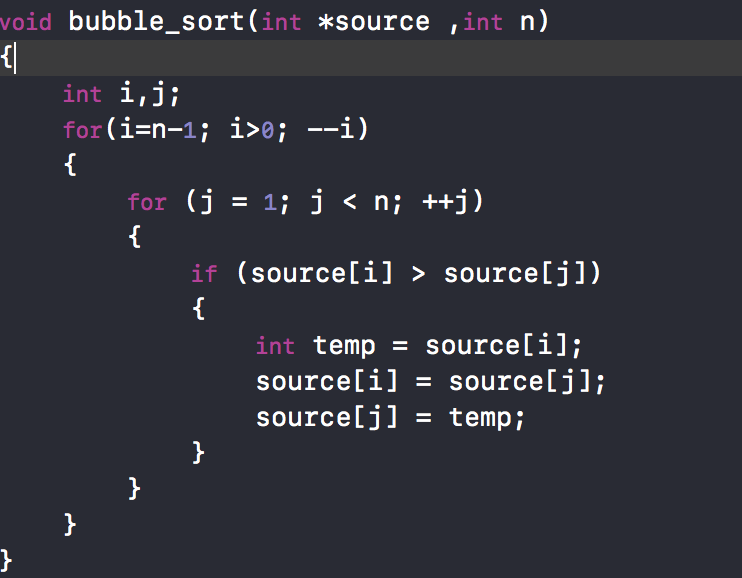
此段代码用于产生随机数组



radix\_sort的原理在于先把数组的每一位取到一个另一个数组里，再对此数组从低位到高位用稳定的排序方法进行排序，我的实现中使用的是冒泡排序，由于冒泡排序的时间复杂度为O（n2），因此时间复杂度为O（n2）。



此函数中，temp[]用于存储未排序的数组，source[]存储排序完成的数组，store[]提供临时的存储空间，排序的时间复杂度为O（n）。



冒泡排序的原理在于从头到尾搜索，每当有逆序的数对时，交换两者，经过两层循环后，排序完成，此排序的时间复杂度为O（n2）。

exp2中的快排与exp1几乎一样，不再赘述。

5、实验结果与分析

exp1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 4 | 32 | 256 | 2048 | 16384 | 131072 |
| insert\_sort | 0.000016 | 0.000023 | 0.000445 | 0.029147 | 1.544183 | 97.492149 |
| heap\_sort | 0.000017 | 0.000034 | 0.000239 | 0.003291 | 0.023611 | 0.232658 |
| quick\_sort | 0.000014 | 0.000024 | 0.000136 | 0.001429 | 0.012603 | 0.137599 |
| merge\_sort | 0.000016 | 0.000037 | 0.000207 | 0.002008 | 0.018859 | 0.201521 |

insert\_sort的维度与其他三者差距较大，故单独列出一个图表。

可以看出，由于选取的点维度差距过于大，散点图的拟合效果十分不理想,因此继续进行处理：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **4** | **32** | **256** | **2048** | **16384** | **131072** |
| **insert\_sort** | **0.000016** | **0.000023** | **0.000445** | **0.029147** | **1.544183** | **97.492149** |
| **heap\_sort** | **0.000017** | **0.000034** | **0.000239** | **0.003291** | **0.023611** | **0.232658** |
| **quick\_sort** | **0.000014** | **0.000024** | **0.000136** | **0.001429** | **0.012603** | **0.137599** |
| **merge\_sort** | **0.000016** | **0.000037** | **0.000207** | **0.002008** | **0.018859** | **0.201521** |
| **time\_of\_insertsort/(n\*n)** | **0.000001** | **2.24609E-08** | **6.79016E-09** | **6.94919E-09** | **5.75253E-09** | **5.67479E-09** |
| **time\_of\_heapsort/(n\*logn)** | **0.000002125** | **2.125E-07** | **1.16699E-07** | **1.46085E-07** | **1.02936E-07** | **1.04414E-07** |
| **time\_of\_quicksort/(n\*logn)** | **0.00000175** | **0.00000015** | **6.64063E-08** | **6.34322E-08** | **5.49447E-08** | **6.17528E-08** |
| **time\_of\_mergesort/(n\*logn)** | **0.000002** | **2.3125E-07** | **1.01074E-07** | **8.91335E-08** | **8.22187E-08** | **9.04402E-08** |

从表格中可以看出当排序的规模扩大以后，time/(时间复杂度）基本上会趋于定值，与课本上的结论基本相同，而规模较小时由于数组交换的时间不固定，出现time/(时间复杂度）不稳定的现象，这是完全可以理解的。对每个算法的执行时间分析，只有insert\_sort函数的运行时间比较高，这也与课本的结果完全符合。

exp2

曲线直接拟合的效果和exp2一样十分不理想，就不给出了，接下来用上面的方法分析算法的时间复杂度：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **4** | **32** | **256** | **2048** | **16384** | **131072** |
| **counting\_sort** | **0.000758** | **0.000773** | **0.000829** | **0.000933** | **0.00178** | **0.003615** |
| **bubble\_sort** | **0.000002** | **0.000006** | **0.000139** | **0.009435** | **0.768796** | **50.128077** |
| **quick\_sort** | **0.000002** | **0.000003** | **0.000026** | **0.000214** | **0.002585** | **0.02142** |
| **radix\_sort** | **0.000011** | **0.000094** | **0.004622** | **0.20954** | **13.297825** | **922.94554** |
| **time\_coutingsort/(n+65535)** | **1.15656E-08** | **1.17895E-08** | **1.26005E-08** | **1.38052E-08** | **2.17288E-08** | **1.83869E-08** |
| **time\_bubblesort/(n\*n)** | **0.000000125** | **5.85938E-09** | **2.12097E-09** | **2.24948E-09** | **2.86399E-09** | **2.91784E-09** |
| **time\_quicksort/(n\*logn)** | **0.00000025** | **1.875E-08** | **1.26953E-08** | **9.49929E-09** | **1.12697E-08** | **9.61304E-09** |
| **time\_radixsort/(n\*n)** | **6.875E-07** | **9.17969E-08** | **7.05261E-08** | **4.99582E-08** | **4.95383E-08** | **5.37225E-08** |

从曲线中可以看出排序规模大时time/(时间复杂度）都为直线，说明实验结果基本正确，从图表中可以看出radix\_sort函数的执行时间最大，这是因为函数中对每个位置都使用了冒泡排序，所以时间会比较大，可以看出time\_radixsort/(n\*n)大约是time\_bubblesort/(n\*n)的十倍，这是因为65535共有5位数字，而每次交换都要将sorce与temp数组都进行交换，这样又会消耗两倍的时间，所以时间是十倍完全符合排序的原理。

6、实验心得

实验的各种排序的难度都不是很大，主要难度在于对于各个边界的调试，实验结果看起来十分漂亮，而且十分符合实验的原理，谢谢老师的用心指导。