

**算法设计与分析**

**实 验 报 告**

学 号： 181110315

学生姓名： 王少博

班 级： 1811103

**2020-10**

**算法设计与分析 实验报告**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 王少博 | | | 院系 | | 软件学院 | | | 学号 | |  |
| 任课教师 | | 王金宝 | | | | | 指导教师 | | 王金宝 | | |
| 实验地点 | | 研究院1号楼507 | | | | | 实验时间 | | 2020年10月12 日 | | |
| 实验名称 | | 使用分治技术的算法设计与实现 | | | | | | | | | |
| 同 组 人 | | 无 | | | | | | | | | |
| 实验内容背景 | | | | | | | | | | | |
| 不同排序算法的运行时间随数据规模增长的情况各不相同，我们将在不同规模的数据上测试各种排序算法，以获取不同排序算法运行时间的增长情况。  Selection问题可以通过先排序，再定位的方式解决，也可以通过课堂上学习的分治（减治）算法直接解决。我们将在不同规模的数据上对比两种解决方式的运行时间。 | | | | | | | | | | | |
| 实验内容（问题，思路，程序，结果） | | | | | | | | | | | |
| 1． 测试数据生成：随机生成不同规模的测试数据，数据类型采用双精度数字（double类型），数据规模为10，102，103，104，105，106，107，108个数字.  2． 实现InsertionSort、MergeSort和QuickSort，测试InsertionSort、MergeSort和QuickSort在不同规模数据上的运行时间。  3. 实现先排序再定位和直接使用分治（减治）的算法分别求解Selection问题，并测试两种方式在不同规模数据上所需的时间。（Selection问题中选择目标的序号可以随机生成，或者指定为数据规模的一半）  4. 绘制线图展示1）不同排序算法在不同规模数据集上的运行时间；2）对比两种解决Selection问题的方法所需的时间。（绘图可以使用GNUPlot、Python或Excel等）。  注意：为了消除随机性，我们为每个规模随机生成10个数据集，随后每个测试在每个数据集上运行10次，一个测试在一个数据规模上运行时间取100次测试的平均值。  本次作业使用jupyter notebook进行编写。   1. **首先是生成数据集：对于8个规模每个都生成10个数据集。**   scales = [10\*\*i for i in range(1,9)]  dataset\_list={}  for i in range(len(scales)):   dataset\_list[i]=np.random.random((10,scales[i]))  scale代表每一个规模，然后数据通过numpy.random.random()生成，均在0-1之间，每一个规模生成10个数据集因此维度是(10, scales[i])   1. **编写排序算法如下：**  * **插入排序：**   def insertionSort(a):   N=len(a)   for i in range(1,N):   for j in reversed(range(1,i+1)):   if a[j]>=a[j-1]:   break   c = a[j]   a[j] = a[j-1]   a[j-1] = c   return a  插入排序思路就像扑克整理牌，从某一个位置往前看这张牌应该插在哪里。由于有两重循环，总体的时间复杂度为   * 归并排序：   def merge(arr, l, m, r):   n1 = m - l + 1   n2 = r- m     L = [0] \* (n1)   R = [0] \* (n2)     for i in range(0 , n1):   L[i] = arr[l + i]     for j in range(0 , n2):   R[j] = arr[m + 1 + j]     i = 0   j = 0   k = l     while i < n1 and j < n2 :   if L[i] <= R[j]:   arr[k] = L[i]   i += 1   else:   arr[k] = R[j]   j += 1   k += 1     while i < n1:   arr[k] = L[i]   i += 1   k += 1     while j < n2:   arr[k] = R[j]   j += 1   k += 1    def mergeSort(arr,l,r):   if l < r:   m = int((l+(r-1))/2)   mergeSort(arr, l, m)   mergeSort(arr, m+1, r)   merge(arr, l, m, r)  归并排序分解待排序的n个元素的序列成各具有n/2个元素的两个子序列。使用归并排序递归的排序两个子序列，合并两个已经排序的子序列为最终的排序结果。由于递归时候两个子序列各需要个时间复杂度，且归并时的时间需要是所以我们有：  由master定理知，   * 快速排序：   def QuikeSort(a):   if a==[]:   return []   else:   small = [x for x in a[1:] if x < a[0]]   big = [x for x in a[1:] if x >= a[0]]   return QuikeSort(small)+[a[0]]+QuikeSort(big)  快速排序的思想简单说就是选择序列里面的某一项作为指标根据这个指标把序列分为小于它的和大于它的（把小的交换到它的左边，大的交换到它的右边），对于这两个子序列再进行类似的操作，最终得到了整个有序序列。我们记这个选择作为标准的这一项的位置为mark，那么它划分出来的两个子序列长度分别是mark和n-mark。同时对于交换操作可以在时间内完成，所以我们有:  对于不同情况下的mark，我们计算T(n)的期望为：  因此，我们有,  令, 通过累加我们有   1. **编写不基于排序的selection算法**   算法描述如下：定义。选一个, 用p 将S 划分为3 个子集合S1, S2, S3:  3 种情况:  若|S1| > k , 则在集合S1中搜索第k小的元素.  否则, 若|S1| + |S2| > k, 则 p 是S中第k小的元素.  否则, 在S3中搜索第(k - |S1| - |S2|)小的元素.  首先设计一种找到p的方法。大概的想法就是每次把序列划分为[N/5]份（最后一份可能不足5但是没有关系）然后求出这些组的中位数，然后把这些中位数在一起求出中位数，以此递归求出最终的中位数。  def calculate\_p(a):   a = np.array(a)   N = a.shape[0]   if N==1:   return a[0]   elif N<5:   if N%2==0:  # print(N)   return 0.5\*(sorted(a)[N//2]+sorted(a)[N//2-1])   else:   return sorted(a)[N//2]   N\_big = N   if not N%5==0:   N\_big = N+5-N%5   fold = N\_big//5   b = np.zeros((fold,5))   b\_list = []   if not N%5==0:   for i in range(fold-1):   b[i]=a[i\*5:(i+1)\*5]   remain = N+5-N\_big   b[fold-1][:remain]=a[N\_big-5:N]   b[fold-1][remain:]=float("inf")   for i in range(fold-1):   b\_list.append(sorted(b[i])[2])   if (-N\_big+N+5)%2==0:   b\_list.append(0.5\*(sorted(b[fold-1][:remain])[remain//2]+sorted(b[fold-1])[remain//2-1]))   else:   b\_list.append(sorted(b[fold-1][:remain])[remain//2])   else:   for i in range(fold):   b[i]=a[i\*5:(i+1)\*5]   b\_list.append(sorted(b[i])[2])  # print(b\_list)   p = calculate\_p(b\_list)   return p  接下来划分S1, S2, S3:  def partition\_p(a):   p = calculate\_p(a)   a\_left,a\_right,a\_equal=[],[],[]   for i in range(len(a)):   if a[i]<p:   a\_left.append(a[i])   elif a[i]==p:   a\_equal.append(a[i])   else:   a\_right.append(a[i])   return a\_left,a\_equal,a\_right  再通过上述算法计算第k小的数：  def selection(a,k):   a\_left,a\_equal,a\_right = partition\_p(a)  # print(a\_left,a\_equal,a\_right)   if len(a\_left)>=k:   return selection(a\_left,k)   elif len(a\_left)+len(a\_equal)>=k:   return a\_equal[0]   else:   k\_ = k-len(a\_left)-len(a\_equal)   return selection(a\_right,k\_)  时间复杂度分析：   1. 分组的时候时间为, 2. 计算中位数时候时间为. 3. 计算这些中位数的中位数的时间为原来规模的1/5，为. 4. 划分三个子集的操作时间为 5. 找到第k个元素的时间复杂度是   经过推导知，   1. **对比各个排序算法计算选出最快的算法**   这里由于时间原因，规模取到了1e7，对于insertion sort只取到了1e4。绘图函数编写如下：  def plot\_fig1():   plt.figure(figsize=(7,5))   plt.plot(range(4),np.log10(insertion\_list))   plt.plot(range(7),np.log10(merge\_list))   plt.plot(range(7),np.log10(quick\_list))   plt.xticks(range(7),['1e'+str(i) for i in range(1,8)])   plt.yticks(range(-5,3),['1e'+str(i) for i in range(-5,3)])   plt.xlabel("Dataset Scale")   plt.ylabel("Average Time for each scale")   plt.title('Compare serveral sort algorithms')   plt.legend(('Insertion','Merge','Quick'))   plt.savefig('sort.jpg',dpi=300)  结果如图所示：    测试函数如下（由于insertion和quick接口一直所以写进了一个函数）：  **这里指定的n为排序次数，根据实验要求默认为10，但是也存在1e1规模下太快导致为0的情况，所以取100次的平均使得非0。**  def test\_insertion\_quick(scale,alg,n=10):   t0 = time.process\_time()   for j in range(n):   for i in range(10):   alg(dataset\_list[scale-1][i].copy())   return (time.process\_time()-t0)/(10\*n)  def test\_merge(scale,alg,n=10):   t0 = time.process\_time()   for j in range(n):   for i in range(10):  # print(dataset\_list[scale-1][i].shape)   alg(dataset\_list[scale-1][i].copy(),0,scales[scale-1]-1)   return (time.process\_time()-t0)/(10\*n)  可以发现，quicksort最快。具体的时间如下：  1. insertion sort:  [2.7623735000000593e-05,  0.001570874670000002,  0.14868630559999998,  15.019524824360001]  2. merge sort:  [2.3348490000012134e-05,  0.0003094620699994266,  0.004308486980000907,  0.05659699914999919,  0.6841428618999998,  7.9797736001500015,  92.31189746439]  3. quick sort:  [2.66940949986747e-05,  0.00019266450000941405,  0.002375610319995758,  0.028415145369999662,  0.3388311810699997,  4.1648118436699955,  58.65553485593002]   1. **quicksort和减治的算法进行对比：**   首先编写selection算法的测试函数：  def test\_selection(scale,n=10):   t0 = time.process\_time()   for j in range(n):   for i in range(10):   selection(dataset\_list[scale-1][i].copy(),k\_list[scale-1])   return (time.process\_time()-t0)/(10\*n)    k\_list = [np.random.randint(scales[i],scales[i+1])//10 for i in range(7)]  k\_list  **输出的结果如下：**  [3.5333858999365476e-05,  0.00021469781000632793,  0.0017644493899933878,  0.01696041311002773,  0.16808913785000187,  1.7034390812499987,  17.119067000019967]  绘图函数编写如下：  def plot\_fig2():   plt.figure(figsize=(7,5))   plt.plot(range(7),np.log10(quick\_list))   plt.plot(range(7),np.log10(selection\_list))   plt.xticks(range(7),['1e'+str(i) for i in range(1,8)])   plt.yticks(range(-5,3),['1e'+str(i) for i in range(-5,3)])   plt.xlabel("Dataset Scale")   plt.ylabel("Average Time for each scale")   plt.title('Compare 2 algorithms')   plt.legend(('Quick','Selection'))   plt.savefig('result.jpg',dpi=300)  结果如下： | | | | | | | | | | | |
| 实验结论（结果分析、遇到的困难和解决方法等） | | | | | | | | | 备注 | |  |
| 分析：  1. 首先，quicksort是三个算法里面最优的算法（从空间和时间来看都是）  2. 我们可以看出，selection在较大规模的数据集上算法会优于基于排序的算法（quick sort）。这里小规模情况下quicksort可能会稍微快一些，猜测原因可能是没有把sort之后取第k个数的操作计算在内。总体来说，不基于排序的算法会更好一些。  3. 实验里面几乎没有遇到困难，唯一的挫折就是python跑数据很慢，跑一次insertion sort就要好几个小时。  4. 除了实验的排序算法之外，我又尝试了python自带的sort算法和numpy里面的排序算法，对于1e8的数据，在10s左右就可以排完。经过查阅得知python里面用到的排序算法是timsort，最好时间复杂度为O(n)。另外numpy和python的部分代码是通过C/C++编写的，本身就会快一些。关于timsort的具体情况如下表所示：    （另外本次作业我是通过jupyter notebook编写的，排版latex已经弄好，放在了附件里面，但是由于实验报告是word所以无法体现这一优势） | | | | | | | | | | | |
| 教师评价 | | | 备注 | |  | | | 得 分 | |  | |
|  | | | | | | | | | | | |