组号: 1

图片包含 游戏机, 画

描述已自动生成

上海大学计算机工程与科学学院

**实 验 报 告**

（数据结构2）

学 期：2021-2022年春季

组 长： 李昀哲

学 号： 20123101

指导教师： 朱能军

成绩评定：

二〇二二年5月28日

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **小组信息** | | | | |
| 登记序号 | 姓名 | 学号 | 贡献比 | 签名 |
| 72 | 李昀哲 | 20123101 | 25％ |  |
| 21 | 唐铭锋 | 20121489 | 25％ |  |
| 20 | 刘沛根 | 20121483 | 25％ |  |
| 22 | 李正宇 | 20121517 | 25％ |  |

|  |  |  |
| --- | --- | --- |
| **实验列表** | | |
| 实验一 | （熟悉上机环境、进度安排、评分制度；分组） | P |
| 实验二 | *有向网的邻接矩阵验证及拓展* | P |
| 实验三 | 无向网的邻接表验证和拓展 | P |
| 实验四 | 查找算法验证及设计 | P |
| 实验五 | 排序算法验证及设计 | P |

实验五

1. **实验题目**

排序算法验证及设计

二、**实验内容**

1. 快速排序算法的改进

对快速排序算法而言，若能合理地选择基准数据元素，使得每次划分后的两个子表中的元素个数尽可能接近，则可以加速排序速度。请设计1-2种基准数据元素选择策略，从而改进教材中实现的快速排序算法，并利用实验结果进行性能比较。

1. DNA排序

字符串的逆序数是指字符串中逆序字符对的数目。例如：字符串“ DAABEC”的逆序数为5，因为D 大于右边4个字符，E大于字符C；又如：字符串“AACEDGG” 的逆序数为1 (只有E与D逆序），它近似已排好序；而字符串“ZWQM”的逆序数为6，它完全逆序。

现要求对DNA字符串（只由A、C、G、T四个字符组成，长度为m）进行分类。分类方法是根据DNA字符串的逆序数从小到大进行排序（自己实现排序算法），逆序数相等的DNA串再按照字符串的字典序从大到小排列。

三、**解决方案**

1、算法设计  
 (a) 对快速排序算法的改进

我们知道，对于快速排序（Quick Sort）而言，由于是个不断递归进行“划分”的过程，因此我们希望能使每次“划分”都较为平均，以此“划分”尽可能少的次数，而“划分”的依据就是选取的基准值（pivot）。一般而言，会简单的将数组中的第一个元素作为基准值，但若涉及到如有序序列等情况，就会使上述中“划分”不均匀的情况出现，极大降低了排序的性能。因此，对快速排序算法的改进聚焦于对基准数据元素的选择。以下将介绍两种选择策略。

选择策略1：取最左、最右边、中心关键字的中值作为基准元素

对于有序的或部分有序的序列来说，以数组的首个元素作为基准值将会进行多次无意义的“划分”，针对这种情况，考虑选择基准元素时，比较数组首个、末个以及中心的元素，选择大小居中的元素作为基准值。

选择策略2：随机选取基准元素

对于一般序列而言，元素大小并没有一定的规律，每一次选择从数组中随机选择位置，将其作为基准值。相比于固定地将首个元素作为基准，会一定程度上提升“划分”的均匀度，从而加速快速排序。

上述二者的优化都是基于快速排序的核心算法，只是在基准值选择上有所区别。

(b) DNA排序

算法流程如下：

1. 输入DNA的个数和等长的长度，维护一个DNA序列的二维数组DNAs；
2. 使用快速排序的思想对DNAs进行排序；
3. 输出排序结果，释放资源。

快速排序对DNAs进行排序时，涉及对DNA序列大小的判断：依据题目要求，在判断DNA序列间大小时，设计了DNACmp（）用于对DNA序列的逆序数（相等时对字典顺序）进行比较。

由于涉及逆序数的计算，又设计了GetInverseNum（）获取每个DNA序列的逆序数，算法核心是用两个游标begin和back分别指向这个DNA序列的头和头的下一个元素，begin依次遍历每一个元素。对于每一个begin，back会遍历它之后的所有元素，若begin指向的元素大于back指向的元素，则逆序数+1.

1. 源程序代码
2. **对快速排序算法的改进**

**获取首个、末个、中心位置元素的中间值算法、快速排序核心算法**

|  |
| --- |
| template<class T>//返回左端，右端，中心元素的中值的位置  int GetMid(T\* begin, int Len) {  if (Len <= 1)  return 0;  int a = begin[0], b = begin[Len / 2], c = begin[Len-1];  int m = std::max(a, b), n = std::max(b, c);  if (m == n)  if (a > c) return 0;  else return Len-1;  else if (m != b && n!= b)  if (a > c) return Len-1;  else return 0;  else  return Len / 2;  }  **//快速排序**  template<class T>  int quick\_sort\_(T\* begin, int Len) {  T\* A = begin;  int mid = GetMid<int>(begin, Len); // 中位  std::swap(A[mid], A[0]); // 把中位数换到第一位  int Tem = A[0]; // 选取比较的基准，其位置也就是初始的坑位  int i = 0, j = Len - 1;  while(i < j) {  while (A[j] >= Tem && i < j)  j--;  if(i < j)  A[i++] = A[j];  while (A[i] <= Tem && i < j)  i++;  if (i < j)  A[j--] = A[i];  }  A[j] = Tem;  return j;  } |

**原始选择策略：以数组首个元素作为基准值的快速排序。**

|  |
| --- |
| #ifndef \_\_QUICK\_SORT\_H\_\_  #define \_\_QUICK\_SORT\_H\_\_  template <class ElemType>  void QuickSort(ElemType elem[], int low, int high, int n)  // 操作结果:对数组elem[low .. high]中的元素进行快速排序  {  ElemType e = elem[low]; // 取枢轴元素  int i = low, j = high;  while (i < j) {  while (i < j && elem[j] >= e) // 使j右边的元素不小于枢轴元素  j--;  if (i < j)  elem[i++] = elem[j];  while (i < j && elem[i] <= e) // 使i左边的元素不大于枢轴元素  i++;  if (i < j)  elem[j--] = elem[i];  }  elem[i] = e;  // cout << "排序区间：" << low << "--" << high << ";中枢位置为：" << i << endl;  // Display(elem, n);  // cout << endl;  if (low < i-1) QuickSort(elem, low, i - 1, n); // 对子表elem[low, i - 1]递归排序  if (i + 1 < high) QuickSort(elem, i + 1, high, n); // 对子表elem[i + 1, high]递归排序  } |

**选择策略1：以数组首个元素作为基准值的快速排序。**

|  |
| --- |
| template<class T>  void QuickSort2(T\* A, int Len) //取三者中关键字居中者作为基准元素的改进方法  {  if (Len <= 1) return;  int k = quick\_sort\_(A, Len);  QuickSort2<int>(A, k);  QuickSort2<int>(&A[k + 1], Len - k - 1);  } |

**选择策略2：随机选择数组位置，将其元素作为基准。**

|  |
| --- |
| template<class T>  void QuickSort3(T\* elem,int low,int high)//随机选取基准元素的改进方法  {  srand((unsigned)time(NULL)); //设置随机数，随机选取枢轴元素  int pos = rand()%(high - low) + low;  T e=elem[pos];  int i = low, j = high;  while (i < j) {  while (i < j && elem[j] >= e) // 使j右边的元素不小于枢轴元素  j--;  if (i < j)  elem[i++] = elem[j];  while (i < j && elem[i] <= e) // 使i左边的元素不大于枢轴元素  i++;  if (i < j)  elem[j--] = elem[i];  }  elem[i] = e;  if (low < i-1) QuickSort3(elem, low, i - 1); // 对子表elem[low, i - 1]递归排序  if (i + 1 < high) QuickSort3(elem, i + 1, high); // 对子表elem[i + 1, high]递归排序  }  #endif |

1. DNA排序

|  |
| --- |
| #include <iostream>  #include <cstring>  using namespace std;  int GetInverseNum(const char\* DNA, int DNA\_len); // 得到DNA的逆序数  /\*\*  \*  \* @param DNA\_1  \* @param DNA\_2  \* @param DNA\_len DNA 的长度  \* @return 如果返回值 < 0，则表示 DNA1 小于 DNA2。  \* 如果返回值 > 0，则表示 DNA1 大于 DNA2。  \* 如果返回值 = 0，则表示 DNA1 等于 DNA2。  \*/  int DNACmp(const char\* DNA\_1,const char\* DNA\_2, int DNA\_len);  void SortDNA(char\*\* DNAs, int DNA\_len, int DNA\_num); // 用快排对DNA进行排序  int main(){  int DNA\_len(0),DNA\_num(0); // DNA的长度和个数  cin >> DNA\_num >> DNA\_len;  char \*\*DNAs = new char\*[DNA\_num];  for(int i(0); i < DNA\_num;++i){ // 读入字符串  char \*DNA = new char[DNA\_len + 1];  cin >> DNA;  DNAs[i] = DNA;  }  SortDNA(DNAs, DNA\_len, DNA\_num);  for(int i(0); i < DNA\_num; ++i) {  cout << DNAs[i] << endl;  delete DNAs[i]; // 顺便逐个删除  }  delete[] DNAs;  return 0;  }  void SortDNA(char\*\* DNAs, int DNA\_len, int DNA\_num){ /// 快排  char \*DNA\_temp(DNAs[0]);  int low(0), high(DNA\_num - 1);  while(low < high){  while(low < high && DNACmp(DNAs[high], DNA\_temp, DNA\_len) >= 0) --high; // 小的放前面  if(low < high) DNAs[low++] = DNAs[high];  while(low < high && DNACmp(DNAs[low], DNA\_temp, DNA\_len) <= 0) ++low; // 大的放后面  if(low < high) DNAs[high--] = DNAs[low];  }  DNAs[low] = DNA\_temp;  if(low - 1 > 0) SortDNA(DNAs, DNA\_len, high - 1);  if(DNA\_num > low + 2) SortDNA(DNAs + low + 1, DNA\_len, DNA\_num - low - 1);  }  int DNACmp(const char\* DNA\_1,const char\* DNA\_2, int DNA\_len){  int inverse\_num\_1(GetInverseNum(DNA\_1, DNA\_len)),  inverse\_num\_2(GetInverseNum(DNA\_2, DNA\_len));  if(inverse\_num\_1 < inverse\_num\_2) return -1; // 若第一个DNA的逆序数小于第二个，  // 返回值小于0  else if(inverse\_num\_1 > inverse\_num\_2) return 1; // 若第一个DNA的逆序数大于第二个，  // 返回值大于0  return -strcmp(DNA\_1,DNA\_2); // 逆序数相等时，返回负的字典比较结果  //（因为字典顺序从大到小排，与逆序数相反）  }  int GetInverseNum(const char\* DNA, int DNA\_len){  int sum(0);  for(int begin(0); begin < DNA\_len - 1; ++begin){ // 循环比较计算出逆序数  for(int back(begin + 1); back < DNA\_len; ++back){  if(DNA[begin] > DNA[back])  ++sum;  }  }  return sum;  } |

1. 实验结果
2. 对快速排序算法的改进

原始选择策略：去数组首个元素作为基准

选择策略1：取首、末、中心三个元素的中间值作为基准元素

选择策略2：随机选取基准元素

表1 数组规模为10,000的不同序列不同策略下的性能测试

|  |  |  |  |
| --- | --- | --- | --- |
| 序列类型 | 原始选择策略 | 选择策略1 | 选择策略2 |
| 完全顺序 | 41.7586 ms | **0.3285 ms** | 1.2481 ms |
| 随机序列 | 0.5721 ms | **0.3764 ms** | 1.1733 ms |

表2 数组规模为100,000的随机序列不同策略下的性能测试

|  |  |  |  |
| --- | --- | --- | --- |
| 序列类型 | 原始选择策略 | 选择策略1 | 选择策略2 |
| 随机序列 | 7.002 ms | **4.001 ms** | 5.001 ms |

1. DNA排序

算法方面的性能和快速排序基本相同，这里仅展示对于DNA序列的正确排序。如图1所示。

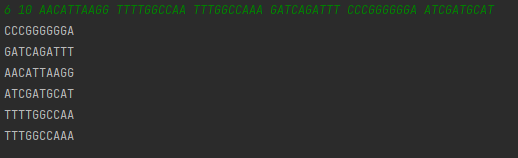


图1 DNA序列排序结果

1. 算法分析

根据优化策略，分别对10,000和100,000数据规模下的序列进行性能测试。如表1 所示，在数据规模为10,000的情况下，完全顺序的序列在原始策略下性能如前文理论相符：需要较多的“划分”次数使时间大幅提升。而在优化选择策略后，排序性能提升较大。针对完全排序序列，优化较为成功。

对于随机序列而言，性能最佳的同样是“选择策略1”，由于选择策略2的思想为随机选择，因此“划分”的均匀性是类正态分布的，可能会出现相比原始策略较慢的情况。但效率的下降是在可接受范围内。

如表2 所示，在数据规模为100,000的情况下，仅考虑随机序列的情况。性能最佳的仍旧是“选择策略1”，在数据规模较大的情况下，随机选择的分布就更为平均，使得“选择策略2”有相较于原始策略更好的性能。

简单总结，对于任意序列，“选择策略1”的表现都较为出色，对于小规模且元素随机的序列，原始策略和“选择策略2”不分伯仲；对于大规模的序列，“选择策略2”相较于原始策略，表现更好。

1. 总结与心得

本次实验是本学期、也是数据结构课程的最后一次小组实验，在本次实验的过程中，尽管小组成员各自在返乡之途中，但我们仍积极开展讨论，明确分工，打好“最后一仗”。

排序算法是数据结构中核心的算法之一，通过实验，我们不仅对排序算法进行了验证、设计优化方法，同时对于一些具体问题如“DNA排序”设计了算法，对排序有了更深入的理解。通过此次实验，我们提高了算法能力，增加了数据结构知识，将数据结构应用于实际。

一学期过的很快，虽然这学期受疫情影响显得有些支离破碎，校园生活也在核酸、抗原、考试、网课中度过，但总体而言，线上课程对于学习的影响并不大，甚至还能借助超星平台，对一些课程录屏有更好的回顾。数据结构的课程行将结束，但数据结构对于我们学计算机的同学将常伴吾身，对它的使用也才刚刚拉开帷幕... ...

**四、分工说明**

李昀哲：算法设计，代码编写，撰写报告

唐铭锋：算法设计，代码编写，ppt制作

刘沛根：算法设计，代码编写，ppt制作

李正宇：算法设计，代码编写，撰写报告