**实 验 报 告 （4 ）**

|  |  |
| --- | --- |
| **实验名称**：排序算法 | **实验地点**：信息楼211、宿舍 |
| **所使用的工具软件及环境：**  windows 11版本 10.0.22631 版本 22631: ；cmd 10.0.22631.4460；powershell  GNU gdb (GDB) 7.6.1,  gcc version 6.3.0 (MinGW.org GCC-6.3.0-1) win32,  Vscode 1.94.2  Ubuntu 24.04 LTS :  gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4) posix,  GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git,  valgrind-3.22.0 | |
| **一、实验目的：**  理解各类排序算法的设计思想，灵活应用排序方法解决实际问题。 | |
| **二、实验内容及要求：**（填写题目内容及输入输出要求）  **1．寻找大富翁。假设给出N个人的个人资产值，请使用堆排序快速找出资产排前M位的大富翁。**  **2．给定数组{48,27,96,48,25,6,90,17,84,62,49,72,17}。请分别用简单插入排序、快速排序和冒泡排序进行排序，输出排序过程中每一趟操作结果，分析比较交换和比较次数以及排序结果是否稳定。** | |
| **三、程序运行结果（说明设计思路，解释使用的数据结构，计算时间复杂度）**  **1）设计思路与时空复杂度、交换/比较次数**  **//由于2项实验内容本质上都是排序，因此设计思路和代码测试将会被一同进行。**   1. 堆排序的核心思想是：利用最大堆（或最小堆）输出堆顶元素，即最大值（或最小），将剩余元素重新生成最大堆（或最小），继续输出堆顶元素直到所有元素都已经被输出。同时，通过将输出后的堆顶元素与最后的一个元素交换的方式，节省程序耗费的内存，只需要使用O(1)的额外空间。   **平均时空复杂度：T(n)=O(Nlog N)，S(n)=O(1)**  **稳定性：不稳定**  **交换/比较次数：堆排序的每次交换都涉及比较，故而二者都为O(Nlog N)**   1. 简单插入排序将待排序的序列分为已经排好序的部分和未排序的部分，初始状态时，已排序序列仅包含第一个元素，未排序序列包含剩余元素。此后将未排序的一个元素插入到已排序序列并使已排序序列有序，重复N-1次插入后，排序完成。   **平均时空复杂度：T(n)=O(N^2)，S(n)=O(1)**  **稳定性：稳定**  **交换/比较次数：简单插入排序在插入时，交换（移动）和比较的次数几乎一致。故二者都为O(N^2)[最坏]**   1. 快速排序，本质上是分治法。每次将1个未排序的元素作为一个基准“主元”（Pivot），将原序列分为2个子序列，其中1个子序列的记录均大于主元，另1个子序列均小于主元，然后递归对左右子序列进行同样的操作。依据中值准则选取主元，并设置问题规模<cutoff时调用简单插入排序来结束递归并完成排序。   **平均时空复杂度：T(n)=O(Nlog N)，S(n)=O(log N)**  **稳定性：不稳定**  **交换/比较次数：快速排序的交换和比较次数取决于选取的基准主元，平均都为O(Nlog N)，最坏都为O(N^2)**  4.冒泡排序，简单来说就是每轮不断将序列内相邻元素进行大小比较并按照特定规则交换（如大于则交换），执行N-1轮或本轮结束却未发生交换时，序列即有序。  **平均时空复杂度：T(n)=O(N^2)，S(n)=O(1)**  **稳定性：稳定**  **交换/比较次数：冒泡排序的交换和比较次数最坏情况下基本是一一对应的，都为O(N^2)**  **其他设计：**   1. 本实验中的算法为了编写方便统一为升序排序，根据题目条件反转/保持原样即可。顺带写了选择排序。 2. 规范化函数声明的写法，使得鼠标悬停在函数名上可以看见该函数的相关提示。 3. 采取模块化编程，将声明写在头文件内，定义写在源文件内，使用#pragma once 预编译指令防止头文件被多次编译，不同功能的函数放在不同的源文件内。 4. 重定向stderr用来生成log文件辅助调试，结合了errno.h。 5. 使用gdb进行断点调试，valgrind进行内存泄露检查。 6. 编写了人性化的输入输出，并且完备了内存回收机制，正常运行时可以完全通过valgrind的各项检查。//不考虑exit（）退出的内存回收。 7. 使用了指针实现的异或交换算法进行试验，并且完成debug顺带在CSDN上写了一篇blog，   <https://blog.csdn.net/weixin_41721338/article/details/144273226?spm=1001.2014.3001.5501>   1. 使用指针实现了快速排序算法，并对比了直接操作指针访问数据，和利用[]变址运算符访问数据的性能差异。[i] 等价于 \*(p+i)，因此理论上变址运算符要比指针访问数据多做一次加法运算（在\*p++/[i++]情况下），n个数据就是n次。实测当数据规模在5e6时开始有较为确定的快慢关系：即连续的10次测试中直接操作指针实现的快速排序速度都要快于利用变址运算符/数组实现的快速排序。当数据规模在3e7时，两者的10次排序耗时平均差距约为100ms[直接操作指针更快] 2. 利用int main(int argc,char \*argv[]) 设定程序参数，使得exp-4.exe -t时隐藏输出并进行基准测试。正常运行时正常输出，不进行基准测试。 3. 利用C语言可变参数va\_list 实现了基准测试函数，能够很方便的调整测试的函数（只要接口统一）。 4. 利用cmd/powershell type/cat和管道符 | 进行数据输入，避免了额外的数据读取代码。E.g.: type test\_8e6.txt | exp-4.exe -t 5. 利用generate\_nums.py生产力5e6、2e7等量级的测试数据，测试了程序在极端情况下的运行能力、稳定性和可靠性。（大量级数据需要指定编译选项 –-stack）   e.g.: gcc -Wl,--stack=268435456 \*.c -o exp-4.exe -fexec-charset=gbk -g  13.由于数据规模较大时人无法短时间内判断排序结果的正确性，故采用在基准测试中采用交叉验证（算法稳定性不影响交叉验证结果），并输出交叉验证结果cross\_check:true/false;  **2、数据结构的描述**  **1） 详情参见//sort.h**  //本程序排序算法   升序 从小到大排列    /\*  @brief  堆排序  @param  要排序的数组  @param  数组长度  @return None  \*/  void heap\_sort(int arr[],const int len);    /\*  @brief 冒泡排序  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void bubble\_sort(int arr[],const int len);    //冒泡排序     数组传参 int arr[]传递数组(指针调用), int \*arr 传递数组的地址  /\*  @brief 选择排序  ——实验不要求 自己之前写着玩的  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void selection\_sort(int arr[],const int len);     //选择排序  /\*  @brief 插入排序  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void insertion\_sort(int arr[],const int len);              //插入排序  /\*  @brief 复原数组(以便再次进行排序)  @param int []   需要排序的数组  @param int []   目标状态  @param int 数组长度  @return None  \*/  void initialization(int arr[],int arr\_cpy[],const int len);       //复原arr数组  /\*  @brief 打印数组  @param int []   要打印的数组  @param int 数组长度  @return None  \*/  void print\_arr(int arr[],const int len);          //输出数组函数  /\*  @brief  逆转数组(用于升序/降序互相转换)  @param int[]    要逆转的数组  @param int  数组长度  @return  None  \*/  void reverse(int arr[],const int len);    /\*  @brief  打印分界线  @param  int     分界线长度  @return None  \*/  void print\_line(const int num);  /\*  @brief  快速排序,指针实现  @param  int[]   要排序的数组  @param  int\*    左端指针  @param  int\*    右端指针  @param  int     数组长度  @return None  \*/  void Qsort\_p(int arr[],int \*left,int \*right,const int len);  void Qsort(int arr[], int Left, int Right,const int len);  /\*  @brief  书本的中值选择算法  @param   int[]  要排序的数组  @param  int 序列左端  @param  int 序列右端  @return int 返回主元  \*/  int median3(int arr[], int left, int right);    /\*  @brief  快速排序 数组版  @param   int[]  要排序的数组  @param  int 数组长度  @return None  \*/  void QuickSort(int arr[], int len);  /\*  @brief  按照中值标准选取主元,并将主元置于倒数第2个位置  @param  int []  要排序的数组  @param  int\*    左端指针  @param  int\*    右端指针  @return None  \*/  void Median(int arr[],int \*left,int \*right);  /\*  @brief  快速排序 指针实现,统一接口  @param  int[]   要排序的数组  @param  int 数组长度  @return None  \*/  void QuickSort\_p(int arr[],const int len);  /\*  @brief  验证排序后的数组和答案是否一致  @param  int[]   排序好的数组  @param  int[]   答案  @param  int 数组长度  @return bool    一致返回true  \*/  bool validation\_arr(int arr[],int answer[],const int len);  **2）//tool.h**  /\*  @brief  不额外利用变量,利用xor就地交换2个数  @param int\* 数1的指针  @param  int\*    数2的指针  @return None  \*/  void xor\_swap(int \*a,int \*b);  /\*  @brief  对函数进行时间基准测试,利用可变参数和 void 对其他类型的兼容  @param  int[]   测试数据  @param  int 测试数据的长度  @param  int 要传入的可变参数总数  @param  void\*(int [],int)   函数指针,传入argc个要进行时间基准测试的函数  @return None  \*/  void bench\_test\_time(int arr[],const int len,const int argc,...);  **3）//max\_heap.h**  #define MAXDATA \_\_INT\_MAX\_\_  /\* 该值应根据具体情况定义为大于堆中所有可能元素的值 \*/  #define ERROR -1 /\* 错误标识应根据具体情况定义为堆中不可能出现的元素值 \*/  typedef int ElementType;  typedef struct HNode \*Heap; /\* 堆的类型定义 \*/  struct HNode {      ElementType \*Data; /\* 存储元素的数组 \*/      int Size;          /\* 堆中当前元素个数 \*/      int Capacity;      /\* 堆的最大容量 \*/  };  typedef Heap MaxHeap; /\* 最大堆 \*/  typedef Heap MinHeap; /\* 最小堆 \*/  /\*  @brief   创建容量为MaxSize的空的最大堆  @param  int  最大堆得最大大小  @param  int  创建的结点个数  @return MaxHeap 最大堆  \*/  MaxHeap CreateHeap(const int MaxSize,const int num );    /\*  @brief 下滤     将H中以H->Data[p]为根的子堆调整为最大堆  @param  MaxHeap 要调整的最大堆  @param  int    根的位置  @param  int 堆内数据的数量  @return None  \*/  void PercDown( MaxHeap H,const int p ,const int N);    /\*  @brief  调整H->Data[]中的元素，使满足最大堆的有序性;假设所有size哥元素已经存入  @param MaxHeap 要调整的最大堆  @return     MaxHeap 调整完成的最大堆  \*/  MaxHeap BuildHeap( MaxHeap H);  /\*  @brief  删除最大堆(释放内存)  @param  MaxHeap     要删除的最大堆  @return None  \*/  void DeleteHeap(MaxHeap H);    **4）//err.h**  /\*  @brief perror(str) 并写入stderr  @param char\* str要写入的错误信息  @return None  \*/  void print\_err(char \*str);  /\*  @brief 重定向stderr至 str.log  @param  char\* 重定向后的文件主名  @return None  \*/  void redirect\_stderr(char \*str) ;  **3、程序代码**  //main.c 23010341 exp-4  #include "sort.h"  #include "max\_heap.h"  #include "err.h"  #include "tool.h"    bool bench\_test=false;      //是否基准测试  int main(int argc,char \*argv[])  {        redirect\_stderr("./exp-4");       //重定向stderr 至./exp-4.log      if(argc>2)      {          print\_err("args set wrong.");          exit(-5);      }      else if(argc==2)  if (strcmp(argv[1],"-t")==0)  bench\_test=true;    //进行基准测试        int len=0;      if(!scanf("%d",&len))       //输入数组长度      {          print\_err("scanf failed.");          exit(-2);      }      int \*arr\_cpy=(int\*)malloc(sizeof(int)\*len);      int \*arr=(int\*)malloc(sizeof(int)\*len);     //分配空间      if(!arr\_cpy||!arr)      {          print\_err("malloc failed! arr\_cpy");       //打印错误          exit(-1);      }      for(int i=0;i<len;i++)      {          if(!scanf("%d",arr+i))      //读入数据          {              print\_err("scanf nums failed.");              exit(-3);          }          \*(arr\_cpy+i)=\*(arr+i);      //   等价于arr\_cpy[i]=arr[i]      }        if(!bench\_test)     //不运行基准测试      {        print\_line(60);     //打印长度为60的分界线      printf("len:%d\n排序前:\n",len);      print\_arr(arr,len);     //打印读取进来的数组      print\_line(60);         //[down] ->降序  [up]->升序      heap\_sort(arr,len);   //堆排序      printf("len:%d\n堆排序:[down](富豪的资产从大到小排序)\n",len);       //降序      reverse(arr,len);      print\_arr(arr,len);         //打印数组(排序结果)      initialization(arr,arr\_cpy,len);    //重置数组      print\_line(60);      bubble\_sort(arr,len);   //冒泡排序      printf("len:%d\n冒泡排序:[up]\n",len);      print\_arr(arr,len);      initialization(arr,arr\_cpy,len);      print\_line(60);      insertion\_sort(arr,len);    //插入排序      printf("len:%d\n简单插入排序:[up]\n",len);      print\_arr(arr,len);      initialization(arr,arr\_cpy,len);      print\_line(60);        QuickSort(arr,len);     //快速排序  指针实现      printf("len:%d\n快速排序(数组实现):[up]\n",len);      print\_arr(arr,len);      initialization(arr,arr\_cpy,len);      print\_line(60);      QuickSort\_p(arr,len);     //快速排序  指针实现      printf("len:%d\n快速排序(指针实现):[up]\n",len);      print\_arr(arr,len);      initialization(arr,arr\_cpy,len);      print\_line(60);      }      else      {       bench\_test\_time(arr,len,3,heap\_sort,QuickSort,QuickSort\_p);        //比较指针快排和数组快排的速度      //bench\_test\_time(arr,len,4,bubble\_sort,heap\_sort,Qsort,insertion\_sort);  //      print\_line(60);      }        /\*            \*/        /\*      selection\_sort(arr,len);    //选择排序      print\_arr(arr,len);      initialization(arr,arr\_cpy,len);      \*/            free(arr\_cpy);      free(arr);      //释放内存      return 0;  }  //headers.h     23010341 exp-4  //类似于 #include <std/bitsc++.h>       集合了所需的所有头文件的头文件  #pragma once  #include <errno.h>  #include <stdio.h>  #include <stdbool.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #include <time.h>  #include <ctype.h>    //err.h 23010341 exp-4  //用于错误输出 23010341杳泽  #pragma once  #include "headers.h"  /\*  @brief perror(str) 并写入stderr  @param char\* str要写入的错误信息  @return None  \*/  void print\_err(char \*str);  /\*  @brief 重定向stderr至 str.log  @param  char\* 重定向后的文件主名  @return None  \*/  void redirect\_stderr(char \*str) ;    //err.c 23010341 exp-4  #include "err.h"    void print\_err(char \*str)        //perror(str)    并写入stderr  {      perror(str);      fprintf(stderr,"errno:%d\n\n",errno) ;      fprintf(stdout,"%s.errno:%d\n",str,errno);      return ;  }  void redirect\_stderr(char \*str)      //重定向stderr 至str.log  {      char location[20];      sprintf(location,"./%s.log",str);       //生成存储文件名      if(freopen(location, "a", stderr) == NULL)fprintf(stdout,"error redirecting stdout\n");      return ;  }    //sort.h  23010341  #pragma once  #include "headers.h"  #include "max\_heap.h"  //本程序排序算法   升序 从小到大排列    /\*  @brief  堆排序  @param  要排序的数组  @param  数组长度  @return None  \*/  void heap\_sort(int arr[],const int len);    /\*  @brief 冒泡排序  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void bubble\_sort(int arr[],const int len);    //冒泡排序     数组传参 int arr[]传递数组(指针调用), int \*arr 传递数组的地址  /\*  @brief 选择排序  ——实验不要求 自己之前写着玩的  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void selection\_sort(int arr[],const int len);     //选择排序  /\*  @brief 插入排序  @param int []   需要排序的数组  @param int 数组长度  @return None  \*/  void insertion\_sort(int arr[],const int len);              //插入排序  /\*  @brief 复原数组(以便再次进行排序)  @param int []   需要排序的数组  @param int []   目标状态  @param int 数组长度  @return None  \*/  void initialization(int arr[],int arr\_cpy[],const int len);       //复原arr数组  /\*  @brief 打印数组  @param int []   要打印的数组  @param int 数组长度  @return None  \*/  void print\_arr(int arr[],const int len);          //输出数组函数  /\*  @brief  逆转数组(用于升序/降序互相转换)  @param int[]    要逆转的数组  @param int  数组长度  @return  None  \*/  void reverse(int arr[],const int len);    /\*  @brief  打印分界线  @param  int     分界线长度  @return None  \*/  void print\_line(const int num);  /\*  @brief  快速排序,指针实现  @param  int[]   要排序的数组  @param  int\*    左端指针  @param  int\*    右端指针  @param  int     数组长度  @return None  \*/  void Qsort\_p(int arr[],int \*left,int \*right,const int len);  void Qsort(int arr[], int Left, int Right,const int len);  /\*  @brief  书本的中值选择算法  @param   int[]  要排序的数组  @param  int 序列左端  @param  int 序列右端  @return int 返回主元  \*/  int median3(int arr[], int left, int right);    /\*  @brief  快速排序 数组版  @param   int[]  要排序的数组  @param  int 数组长度  @return None  \*/  void QuickSort(int arr[], int len);  /\*  @brief  按照中值标准选取主元,并将主元置于倒数第2个位置  @param  int []  要排序的数组  @param  int\*    左端指针  @param  int\*    右端指针  @return None  \*/  void Median(int arr[],int \*left,int \*right);  /\*  @brief  快速排序 指针实现,统一接口  @param  int[]   要排序的数组  @param  int 数组长度  @return None  \*/  void QuickSort\_p(int arr[],const int len);  /\*  @brief  验证排序后的数组和答案是否一致  @param  int[]   排序好的数组  @param  int[]   答案  @param  int 数组长度  @return bool    一致返回true  \*/  bool validation\_arr(int arr[],int answer[],const int len);    // sort.c 23010341  #include "sort.h"  #include "tool.h"  #include "err.h"  extern bool bench\_test; // 是否运行基准测试  void heap\_sort(int arr[], const int len)  {      MaxHeap H;      const int MaxSize =(int)2e8; // 最大堆创建      和gcc编译器相关,比较大的时候,需要指定编译参数 --stack      if (MaxSize < len)      {          print\_err("Heap\_sort maxsize too small. sort.c");          exit(-4);      }      H = CreateHeap(MaxSize, len);      for (int i = 0; i < len; i++)      {          H->Data[i] = arr[i];      }      H = BuildHeap(H); // 调整最大堆      for (int i = len - 1; i > 0; i--)      {          xor\_swap(&H->Data[0], &H->Data[i]); // swap          PercDown(H, 0, i);          if (!bench\_test)          {              for (int i = 0; i < len; i++)              {                  arr[i] = H->Data[i]; // 拷贝最大堆排序完的数据到arr内              }              print\_arr(arr, len); // 打印过程          }      }      for (int i = 0; i < len; i++)      {          arr[i] = H->Data[i]; // 拷贝最大堆排序完的数据到arr内      }      DeleteHeap(H); // 释放最大堆      return;  }  void bubble\_sort(int arr[], const int len)  {      // int temp=0;      for (int i = 0; i < len - 1; i++)      {          bool swaped = false;          for (int k = 0; k < len - 1; k++)          {              if (arr[k] > arr[k + 1])              {                  xor\_swap(&arr[k + 1], &arr[k]);                  /\*                  temp=arr[k+1];                  arr[k+1]=arr[k];                  arr[k]=temp;                  \*/              }              swaped = true;          }          if (!bench\_test)              print\_arr(arr, len); // 打印过程          if (!swaped)              break; // 无交换 说明已经有序 提前结束      }      return;  }  void selection\_sort(int arr[], const int len)  {      int end = 0, min = 0, temp = 0, i\_temp = 0; // 已排序序列末尾,临时变量      while (end < len)      {          min = arr[end];          for (int i = end; i < len; i++)          {              if (arr[i] < min)              {                  min = arr[i];                  i\_temp = i;              }          }          // 交换          if (i\_temp != -1) // 如果有更小的数才交换          {              temp = arr[end];              arr[end] = arr[i\_temp];              arr[i\_temp] = temp;          }          end++;          i\_temp = -1;          if (!bench\_test)              print\_arr(arr, len); // 打印过程      }      return;  }  void insertion\_sort(int arr[], const int len)  {      for (int i = 1, k = i, temp = 0; i < len; i++) // 从第2个元素开始      {          temp = arr[i];          for (k = i; k > 0 && arr[k - 1] >= temp; k--)          {              arr[k] = arr[k - 1];          }          arr[k] = temp;          if (!bench\_test)              print\_arr(arr, len); // 打印过程      }      return;  }  void initialization(int arr[], int arr\_cpy[], const int len)  {      for (int i = 0; i < len; i++)      {          arr[i] = arr\_cpy[i];      }      return;  }  void print\_arr(int arr[], const int len)  {      for (int i = 0; i < len; i++)      {          printf("%d%c", arr[i], ',');      }      printf("\n");      return;  }  void reverse(int arr[], const int len)  {      for (int \*h = arr, \*p = arr + len - 1; h <= p; h++, p--) // h 头指针 p 尾指针      {          xor\_swap(h, p); // 交换2个数组      }      return;  }  void print\_line(const int num)  {      for (int i = 0; i < num; i++)      {          printf("-");      }      printf("\n");      return;  }  void Qsort(int arr[], int Left, int Right,const int len)  {      int Pivot=0, Cutoff=3, Low=0, High=0;      if(Cutoff<=Right-Left)      {      Median(arr, &arr[Left], &arr[Right]);      Pivot=arr[Right-1];        Low = Left;      High = Right - 1;      while (1)      {          while (arr[++Low] < Pivot);          while (arr[--High] > Pivot);          if (Low < High)              xor\_swap(&arr[Low], &arr[High]);          else              break;      }      xor\_swap(&arr[Low], &arr[Right - 1]);      if (!bench\_test)    print\_arr(arr, len); // 非基准测试下打印过程        Qsort(arr, Left, Low - 1,len);      Qsort(arr, Low + 1, Right,len);      }      else      {            insertion\_sort(arr+Left,Right-Left+1);      }  }  void QuickSort(int arr[], int len)  {      Qsort(arr, 0, len - 1,len);  }  int median3(int arr[], int left, int right)  {      int center = (left + right) / 2;      if (arr[left] > arr[center])          xor\_swap(&arr[left], &arr[center]);      if (arr[left] > arr[right])          xor\_swap(&arr[left], &arr[right]);      if (arr[center] > arr[right])          xor\_swap(&arr[center], &arr[right]);      xor\_swap(&arr[center], &arr[right - 1]);      return arr[right - 1];  }  void Qsort\_p(int arr[], int \*left, int \*right, const int len)  {      const int cutoff = 3;      if (cutoff <= right - left) // 调用快排      {          Median(arr, left, right);           // 按照中值标准选取主元,并将主元置于倒数第2个位置          int \*pivot = right - 1;             // 主元          int \*low = left, \*high = right - 1; // 左端起始和右端起始          while (1)                           // 因为 arr[i]   相当于\*(arr+i)      因此直接操作指针在快排时会比使用变址运算符[]更快,每次都减少一次加法运算          {              /\*              while (\*low < \*pivot)                  low++;              while (\*high > \*pivot)                  high--;                  \*/  //这样写运气好的话可能会进死循环    当\*pivot=\*high=\*low                  // test\_200\_d.txt的数据                  // 显然\*++low  和\*low++ 这2种写法也行不通)              while(\*++low<\*pivot);              while(\*--high>\*pivot) ;              if (low < high)                  xor\_swap(low, high); // 交换2个数              else                  break;          }          xor\_swap(low, pivot);          if (!bench\_test)              print\_arr(arr, len); // 非基准测试下打印过程          Qsort\_p(arr, left, low - 1, len);  // 递归解决左边          Qsort\_p(arr, low + 1, right, len); // 递归解决右边      }      else      {          insertion\_sort(left, right - left + 1); // 规模过小调用插入排序      }      return;  }  void QuickSort\_p(int arr[], const int len)  {      Qsort\_p(arr, arr, arr + len - 1, len);      return;  }  void Median(int arr[], int \*left, int \*right)  {      const int middle = (left - arr + right - arr) / 2;      if (\*left > arr[middle])          xor\_swap(left, &arr[middle]);      if (\*left > \*right)          xor\_swap(left, right);      if (arr[middle] > \*right)          xor\_swap(&arr[middle], right);      xor\_swap(&arr[middle], right - 1); // 问题代码      return;  }  bool validation\_arr(int arr[], int answer[], const int len)  {      // 本例中的数组长度应该都是一致的      for (int i = 0; i < len; i++)      {          if (arr[i] != answer[i])              return false;      }      return true;  }  //tool.h 23010341   exp-4  #pragma once  #include "headers.h"  /\*  @brief  不额外利用变量,利用xor就地交换2个数  @param int\* 数1的指针  @param  int\*    数2的指针  @return None  \*/  void xor\_swap(int \*a,int \*b);  /\*  @brief  对函数进行时间基准测试,利用可变参数和 void 对其他类型的兼容  @param  int[]   测试数据  @param  int 测试数据的长度  @param  int 要传入的可变参数总数  @param  void\*(int [],int)   函数指针,传入argc个要进行时间基准测试的函数  @return None  \*/  void bench\_test\_time(int arr[],const int len,const int argc,...);  //tool.c    23010341    exp-4  #include "tool.h"  #include "sort.h"  #include <stdarg.h>     //实现bench\_test\_time的可变参数  void xor\_swap(int \*a,int \*b)  {      if(a!=b)    //地址相同不需要交换 ,同一个地址自我异或会产生0      {      \*a^=\*b;      \*b^=\*a;      \*a^=\*b;      }      return;  }    void bench\_test\_time(int arr[],const int len,const int argc,...)  {      int arr\_cpy[len],arr\_answer[len];      for(int i=0;i<len;i++)  arr\_cpy[i]=arr[i];      //      va\_list valist;      va\_start(valist,argc);      //为argc个参数初始化valist      void (\*f\_p[argc])(int[],int);  //函数指针        clock\_t start,stop;     //函数开始时间和结束时间      long int duration=0;        //函数运行所用时间      bool correct=true;      int times[argc][10];        //10次时间      for(int i=0;i<argc;i++)      {          f\_p[i]= va\_arg(valist, void (\*)(int[], int)); // 获取下一个函数指针      }      for (int t = 0; t < 10; t++) // 测试10次      {          correct=true;          for (int i = 0; i < argc; i++)          {                start = clock();              f\_p[i](arr, len);              stop = clock();                      // 停止计时              duration = (stop - start); /// CLK\_TCK;            //linux下 应该是 CLOCKS\_PER\_SEC              //   /CLK\_TCK 则转换为s  这里保留ms              printf("函数%d  所用时间:%7ld\tms\n", i + 1, duration);              times[i][t]=duration;              if (i == 0) // 复制第一个算法的答案              {                  for (int k = 0; k < len; k++)                  {                      arr\_answer[k] = arr[k]; // 复制答案                  }              }              else                  correct &= validation\_arr(arr, arr\_answer, len); // 交叉验证              initialization(arr, arr\_cpy, len);                   // 重置数组            }          correct?printf("cross\_check:true\n"):printf("corss\_check:false\n");   //打印交叉验证结果          print\_line(60);          va\_end(valist);     //清理为valist保留的内存      }      for(int i=0;i<3;i++)      {          double avg=0,sum=0;          for(int j=0;j<10;j++)          {              sum+=times[i][j];          }          avg=sum/10.0;          printf("函数%d 10次平均耗时:%7.1lf\tms\n",i+1,avg);      }          return;  }    //max\_heap.h  23010341  exp-4  #pragma once  #include<stdio.h>  #include<stdlib.h>  #define MAXDATA \_\_INT\_MAX\_\_  /\* 该值应根据具体情况定义为大于堆中所有可能元素的值 \*/  #define ERROR -1 /\* 错误标识应根据具体情况定义为堆中不可能出现的元素值 \*/  typedef int ElementType;  typedef struct HNode \*Heap; /\* 堆的类型定义 \*/  struct HNode {      ElementType \*Data; /\* 存储元素的数组 \*/      int Size;          /\* 堆中当前元素个数 \*/      int Capacity;      /\* 堆的最大容量 \*/  };  typedef Heap MaxHeap; /\* 最大堆 \*/  typedef Heap MinHeap; /\* 最小堆 \*/  /\*  @brief   创建容量为MaxSize的空的最大堆  @param  int  最大堆得最大大小  @param  int  创建的结点个数  @return MaxHeap 最大堆  \*/  MaxHeap CreateHeap(const int MaxSize,const int num );    /\*  @brief 下滤     将H中以H->Data[p]为根的子堆调整为最大堆  @param  MaxHeap 要调整的最大堆  @param  int    根的位置  @param  int 堆内数据的数量  @return None  \*/  void PercDown( MaxHeap H,const int p ,const int N);    /\*  @brief  调整H->Data[]中的元素，使满足最大堆的有序性;假设所有size哥元素已经存入  @param MaxHeap 要调整的最大堆  @return     MaxHeap 调整完成的最大堆  \*/  MaxHeap BuildHeap( MaxHeap H);  /\*  @brief  删除最大堆(释放内存)  @param  MaxHeap     要删除的最大堆  @return None  \*/  void DeleteHeap(MaxHeap H);    //max\_heap.c  23010341 exp-4  #include "max\_heap.h"  MaxHeap CreateHeap(const int MaxSize,const int num )  {      int i;      MaxHeap H = (MaxHeap)malloc(sizeof(struct HNode));      H->Data = (ElementType \*)malloc(MaxSize\*sizeof(ElementType));      if(!H|!H->Data)      {          printf("heap malloc failed.\n");          exit(-8);      }      H->Size = num;      H->Capacity = MaxSize;      //H->Data[0] = MAXDATA; /\* 定义"哨兵"为大于堆中所有可能元素的值\*/      //无哨兵      return H;  }  void PercDown( MaxHeap H,const int p,const int N)       //无哨兵的perc Down 需要调整 yaoze--    12.04  { /\* 下滤：将H中以H->Data[p]为根的子堆调整为最大堆 \*/  //本程序的Heap从[0]开始存储数据 因此2i+1为左孩子 2i+2 为右孩子  父节点 floor((i-1)/2)      int Parent, Child;      ElementType X;      X = H->Data[p]; /\* 取出根结点存放的值 \*/      for( Parent=p; Parent\*2+1<N; Parent=Child ) {          Child = Parent \* 2+1;          if( (Child!=N-1) && (H->Data[Child]<H->Data[Child+1]) )              Child++;  /\* Child指向左右子结点的较大者 \*/          if( X >= H->Data[Child] ) break; /\* 找到了合适位置 \*/          else  /\* 下滤X \*/              H->Data[Parent] = H->Data[Child];      }      H->Data[Parent] = X;  }  MaxHeap BuildHeap( MaxHeap H )  { /\* 调整H->Data[]中的元素，使满足最大堆的有序性  \*/    /\* 这里假设所有H->Size个元素已经存在H->Data[]中 \*/      /\* 从最后一个结点的父节点开始，到根结点1 \*/      for(int i = H->Size/2 -1; i>=0; i-- )          PercDown( H, i,H->Size );      return H;  }  void DeleteHeap(MaxHeap H)  {      free(H->Data);      //释放最大堆内的数据      free(H);            //释放最大堆      return ;  }  #generate\_nums.py  #只是用来产生测试数据  import random  nums=int(3e7)  #临时生产随机数序列的脚本,不需if \_\_name\_\_=="\_\_main\_\_"  # 生成一个长度为20000的随机整数列表，每个整数范围从1到10000  random\_integers = [str(random.randint(1, 100000)) for \_ in range(nums)]  # 将列表中的元素用空格连接成一个字符串  random\_sequence = ' '.join(random\_integers)  # 打印结果  print(nums)  print(random\_sequence)   1. **运行界面截图（结果和说明）**   **1、小规模数据，按照要求输出每一步的过程。**    **//上图，20个元素的原始数据**    **//上图，20个元素的堆排序，默认升序，后执行了反转操作（富豪资产从大到小）前M位一个for循环即可实现。**    **//上图，20个元素的升序冒泡排序**    **//上图，20个元素的升序简单插入排序**  **//上图，20个元素的升序快速排序**  **//程序内存检查概要——无泄漏（虽有warning，因为开了很大的空间但相当一部分没有使用[为后续大规模数据测试]）**  **2、大规模数据，进行10次排序，记录不同算法其10次耗时和平均耗时，交叉验证**    **//上图，数据规模过大6e7时，堆排序申请空间malloc失败。堆排序设定的Maxsize大于6e7，但可能是数据小的时候编译器优化了。 --stack=1073741824 约为1024MB的栈**    **//上图，数据规模为3e7时的排序 10次排序耗时**  **//函数1 堆排序（用于交叉验证）**  **//函数2 []实现的快速排序**  **//函数3 直接操作指针实现的快速排序**    **//上图，数据规模为3e7时的排序 平均10次耗时**  **//函数1 堆排序（用于交叉验证）**  **//函数2 []实现的快速排序**  **//函数3 直接操作指针实现的快速排序**  可以看出，在3e7数据规模下，直接操作指针实现的快速排序比[]实现的快速排序10次平均快约100ms；C语言加法运算1e8（1亿）次耗时约为1s；固3e7数据规模下时间仅节省100ms是合理的。每次递归会减小问题的规模，跳过主元的判断，减少判断次数，因此3e7数据规模下节省的加法运算次数少于3e7 次，因此节省的时间少于300ms[也不排除是编译器和运行环境所致]  由于时间限制和对编译原理、优化等底层知识的不了解，最大只测试到了3e7的数据规模。    **////上图，数据规模为8e6时的排序 10次排序耗时**  **//函数1 堆排序（用于交叉验证）**  **//函数2 []实现的快速排序**  **//函数3 直接操作指针实现的快速排序**    **//上图，数据规模为8e6时的排序 10次排序平均耗时**  **//函数1 堆排序（用于交叉验证）**  **//函数2 []实现的快速排序**  **//函数3 直接操作指针实现的快速排序**  8e6数据规模下节省加法运算带来的性能提升不显著，但直接操作指针的快速排序在10次测试中有9次都更快/持平。某种程度上也能说明确实节省了加法运算。由于基准测试函数基于clock()和clock\_t变量实现，因此在性能提升不显著时，CPU时钟主频的震荡会对计时造成一定影响。    //上图，2e5数据规模下，4种排序算法前8次排序耗时[一张图装不下10次]  //函数1 冒泡排序 //函数更改可通过更改main.c:99附近实现  //函数2 堆排序  //函数3 快速排序  //函数4 简单插入排序    //上图，2e5数据规模下，4种排序算法后2次排序耗时[一张图装不下10次]和10次平均耗时  //函数1 冒泡排序 //函数更改可通过更改main.c:99附近实现  //函数2 堆排序  //函数3 快速排序  //函数4 简单插入排序  可以看出，数据规模较大时，4种排序算法的效率/速度:  **快速排序（cutoff=3）> 堆排序 > 简单插入排序 > 冒泡排序**  **3、题目2给定的数组排序及其过程**    //**题目2给定的数组排序及其过程**    **//题目2给定的数组排序及其过程 中间零散值 为快排调用简单插入排序。** | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2024年 月 日