作业 HW6 实验报告

姓名：范潇　学号：2254298　日期：2024 年 3 月 11 日

1. 涉及数据结构和相关背景

现实中的许多应用需要对数据进行排序。同时，许多算法中也包含了“排序”这一子步骤。因此，研究排序算法十分重要。有许多排序算法，评价它们的质量好坏主要依据一下几个指标：时间复杂度，所需附加存储，复杂性。

排序算法的时间复杂度主要体现在所需的平均比较次数和平均移动次序上。如果一个排序算法在排序过程中，会直接将待排序的最大（小）元素调整至它的最终位置，则称该排序算法具有部分排序功能。如果一个排序算法不会改变排序依据——关键字——相同的数据的相对位置，则称该排序算法具有稳定性。

2. 实验内容

2.1. 排序

问题描述：

排序算法分为简单排序（时间复杂度为 $O(n^2)$）和高效排序（时间复杂度为 $O(n\log n)$）。

请自行随机生成不同规模的数据（例如 10，100，1K，10K，100K，1M，10K 正序，10K 逆序），用不同的排序算法（快速排序，归并排序，堆排序，选择排序，冒泡排序，直接插入排序，希尔排序）分别对这些数据进行测试，输出运行时间，并总结各种排序算法的特点、时间复杂度、空间复杂度，以及是否是稳定排序和部分排序。

输入要求：

第 1 行一个正整数 n，表示元素个数。第 2 行 n 个整数，用空格分割。$1 \leq n \leq 100000$

输出要求：

从小到大排序后的结果，以空格分割。

2.1.1. 数据结构设计

```
vector<int> nums;
vector<int> gaps;//用于希尔排序
```

2.1.2. 功能描述

Function Insert-Sort-with-Gap(nums,a_0,gap)

// a_0 为公差为 gap 的子序列的第一个元素。当 $a_0 = 1, gap = 1$ 时，便是直接插入排序

```
for i=a_0 + gap to nums.length by gap do
    j = i-gap
    key = nums[i]
    while j ≥ a_0 and nums[j]>key do
        nums[j+gap]=nums[j]
        j = j-gap
    end while
    nums[j+gap]=key
end for
```

Algorithm 2.1: Shell-Sort(nums,gaps)

```
foreach gap in gaps do
    for i = 0 to gap-1 do Select-Sort-with-Gap (nums,i,gap)
end foreach
```

Algorithm 2.2: Bubble-Sort(nums)

```
1 for i = 1 to nums.length-1 do
2     change = 0 for j = 1 to nums.length-1 do
3         if nums[j+1]<nums[j] then Swap (nums[j],nums[j+1])
4         change = 1
5     end for
6     if change == 0 then break
7 end for
```

Algorithm 2.3: Select-Sort(nums)

```
1 for i = 1 to nums.length-1 do
2     min = i
3     for j = i+1 to nums.length do
4         if nums[j]<nums[min] then min = j
5     end for
6     if min ≠ i then Swap (nums[min],nums[i])
7 end for
```

2.1.3. 调试分析

最初测试归并排序和快速排序时，发现均有 5 个样例出现超时的现象，不符合预期。经过排查后也没有发现逻辑上的问题，最终发现是因为递归函数的返回值是一整个 vector, 但是并没有以引用的形式返回，导致需要大量额外的内存拷贝。将返回变量改为引用形式后，测试结果便正常了。

2.1.4. 总结和体会

使用希尔排序时，当 gaps 设置为 5，3，1 和 4，2，1 时，样例 7 均无法通过；当设置为 8，4，2，1 时，能通过所有样例。

使用插入排序时，无法通过样例 5，7，9 。

使用冒泡排序时，当使用 change 变量判断排序是否完成时，无法通过样例 5，7，8，9 ；不使用时，则无法通过样例 5，6，7，8，9 。

使用选择排序时，无法通过样例 5，6，7，8，9.

使用归并排序和堆排序时，可以通过所有样例。

使用以末端元素作为枢纽的快速排序时，无法通过样例 6，7，8.

由此可以推断出:

- 样例 7 逆序对较多
- 样例 6 的原始序列大部分已经排序完成
- 样例 5，6，7，8，9 待排序元素较多
- 样例 6，7，8 数据的分布会使得快速排序退化。

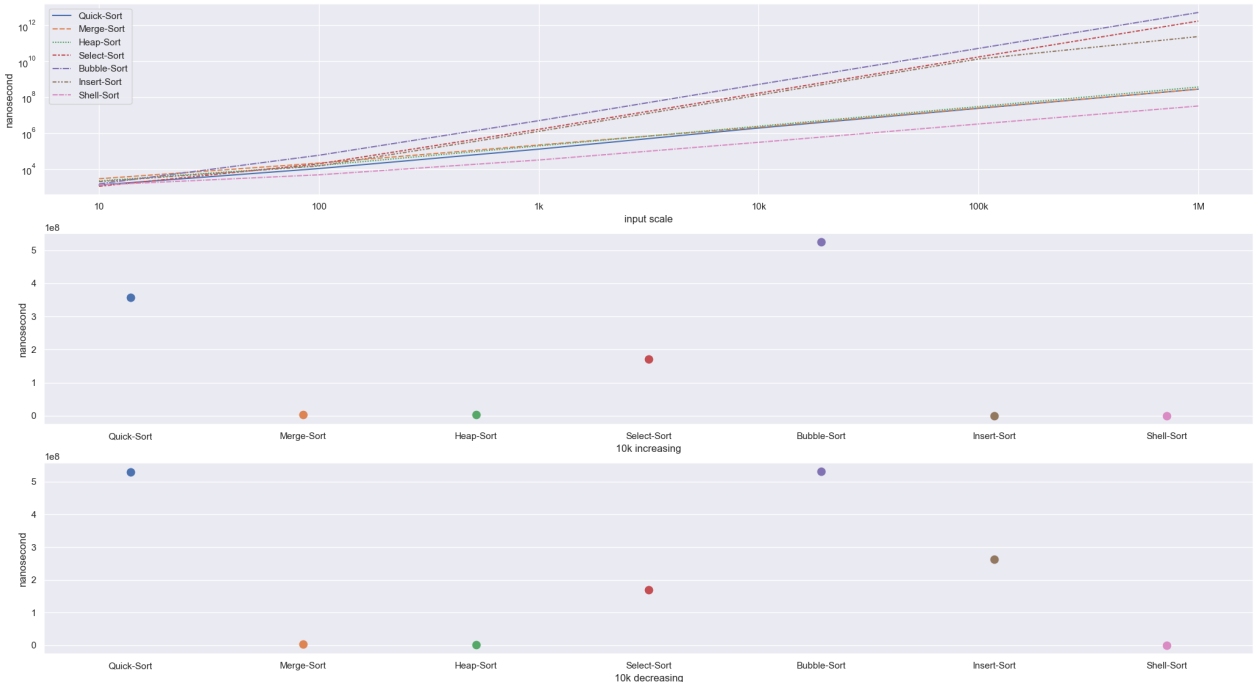

图 1: 各算法运行时间

nanosecond	10	100	1k	10k	100k	1M
Quick-Sort	1300	10700	129000	1939500	24167500	283930100
Merge-Sort	2900	21600	219600	2172600	25443000	283943100
Heap-Sort	2200	15800	195800	2443000	30363500	372333700
Select-Sort	1100	19700	1641100	168638300	17555834200	1754645573400
Bubble-Sort	1500	58800	4971400	520298700	52657170700	5270084636700
Insert-Sort	2000	15000	1268000	131088000	13537059000	240677606000
Shell-Sort	1300	4800	32100	310000	3250400	32636200

nanosecond	10k increaing	10k decreasing
Quick-Sort	356991600	529755200
Merge-Sort	2764800	3034900
Heap-Sort	2458500	2496800
Select-Sort	170450800	168860200
Bubble-Sort	525029500	531327100
Insert-Sort	91000	263311000
Shell-Sort	317800	612800

表 1: 各算法运行时间

	比较次数	移动次数	附加存储空间	稳定排序	部分排序
快速排序	$\Omega(n\lg n), O(n^2)$	$\Omega(n\lg n), O(n^2)$	$\Omega(\lg n), O(n)$	$\times$	$\times$
归并排序	$\Theta(n\lg n)$	$\Theta(n\lg n)$	$\Theta(n)$	$\surd$	$\times$
堆排序	$\Theta(n\lg n)$	$\Theta(n\lg n)$	$\Theta(1)$	$\times$	$\surd$
选择排序	$\Theta(n^2)$	$O(n)$	$\Theta(1)$	$\times$	$\surd$
冒泡排序	$\Omega(n), O(n^2)$	$O(n^2)$	$\Theta(1)$	$\surd$	$\surd$
直接插入排序	$\Omega(n), O(n^2)$	$O(n^2)$	$\Theta(1)$	$\surd$	$\times$
希尔排序	$\Omega(n), O(n^2)$	$O(n^2)$	$\Theta(1)$	$\times$	$\times$

表 2: 各算法特点总结

- 快速排序：是高效排序，并且平均而言性能更加突出，但是遇到某些输入时会退化为简单排序。同时由于是采用递归实现，会占用一定的空间
- 归并排序：是高效排序，且性能不受输入数据的初始状态影响，具有稳定性，但是需要较多的附加存储空间
- 堆排序：是高效排序，且性能不受输入数据的初始状态影响，且具有部分排序功能
- 选择排序：直观易懂，具有部分排序功能，性能较差，但是移动次数较少
- 冒泡排序：直观易懂，具有稳定性，部分排序功能，但性能较差
- 直接插入排序：直观易懂，具有稳定性，但性能较差
- 希尔排序：性能受 gap 的取值影响较大

2.2. 求逆序对数

问题描述：请求出整数序列 A 的所有逆序对个数。

输入要求：

输入包含多组测试数据，每组测试数据有两行。第一行为整数 $N(1 \leq N \leq 20000)$，当输入 0 时结束；第二行为 N 个整数，表示长为 N 的整数序列。

输出要求：

每组数据对应一行，输出逆序对的个数。

2.2.1. 数据结构设计

```
typedef struct {
    int data[MAXN] ;//整数序列
    int length ;//整数序列长度
}array ;
```

2.2.2. 功能描述

Function Merge(A,r,l)

```
mid = ⌊(r+l)/2⌋
n1 = mid - l + 1
n2 = r - mid
创建大小分别为 n1, n2 的数组 L, R
for i=0 to n1-1 do L[i]=A[l+i]
for i=0 to n2-1 do R[i]=A[mid+1+i]
m = 0
n = 0
counter = 0
for i=0 to n1 + n2 − 1 do // 合并子序列
    if (L[m] ≤R[n] and m <n1) or (n ≥ n2) then
        A[l+i] = L[m]
        m = m + 1
    else
        A[l+i] = R[n]
        counter = counter + n1 + n - i// 需要移动的“步数”
        n = n + 1
    end if
end for
return counter
```

Function CountInversion(A,r,l)

```
// A 为给定整数序列
// 最终解答调用 CountInversion (A,0,A.length-1)
if r == l then return 0
mid = ⌊(r+l)/2⌋
return CountInversion (A,l,mid) + CountInversion (A,mid+1,r) + Merge (A,l,r)
```

2.2.3. 调试分析

本题采用输出关键变量进行调试。选用的关键变量是 `Merge` () 中涉及到的数组部分以及对应的 $counter$。

在调试的过程中，遇到了多次相同输入，但是输出不同的情况，因此判断是内存出现问题。查看数组部分的输出信息，发现在运行过程中数组内部出现了异常元素。经过排查后发现是“合并子序列”步骤中，当一个子序列以及全部转移完毕时，仍有可能继续转移，从而产生内存溢出，加上 $m < n_1, n \geq n_2$ 条件后便恢复正常。

2.2.4. 总结和体会

我一开始使用折半插入排序，但是最终有两个样例由于超时而无法通过，这是因为折半插入排序移动元素的平均时间复杂度仍为 $O(n^2)$。最后我在归并排序的基础上进行调整，时间复杂度维持在了 $\Theta(n \lg n)$。使用该方法时，我仍然有两个样例没有通过，原因是内存使用过多，经过排查，是因为只申请了内存，而没有释放。在调试的过程中，我也体会到了处理边界情况的重要性，如果能够合理设置哨兵，例如在数组 L, R 结尾设置两个哨兵，便可以在保持程序简洁的同时确保正确性。

2.3. 三数之和

问题描述：

给你一个整数数组 $nums$ ，判断是否存在三元组 $[nums[i], nums[j], nums[k]]$ 满足 $i \neq j, i \neq k$ 且 $j \neq k$ ，同时还满足 $nums[i] + nums[j] + nums[k] == 0$ 。请你返回所有和为 0 且不重复的三元组，每个三元组占一行。

输入要求：

2 行。第 1 行一个整数，表示数组元素个数；第 2 行输入一组整数，中间以空格分隔。

输出要求：

输出所有和为 0 的三元组，每个三元组一行，中间以空格分隔。对于每一个三元组，你需要按从小到大的顺序依次返回三个元素；对于所有三元组，你需要按三元组中最小元素从小到大的顺序依次打印每一组三元组。

2.3.1. 数据结构设计

```
7 int nums[MAXN];
8 int sums[MAXN]; //存储所有2组合之和
```

2.3.2. 功能描述

Function Heapify(nums,i,size)

```
1 max = i
2 if 2i ≤ size and nums[2i]>nums[max] then max = 2i
3 if 2i + 1 ≤ size and nums[2i+1]>nums[max] then max = 2i+1
4 if max ≠ i then
5 |   Swap (nums[max],nums[i])
6 |   Heapify (nums,max,size)
7 end if
```

Function Build-Heap(nums)

```
1 for i = ⌊n/2⌋ downto 1 do Heapify (nums,i)
```

Algorithm 2.4: Heap-Sort(nums)

```
1 Build-Heap (nums)
2 for i = nums.length downto 2 do
3 |   Swap (nums[1],nums[i])
4 |   Heapify (nums,1,i-1)
5 end for
```

Function Triple-Sum(nums)

```
1  Heap-Sort (nums)
2  i = 1
3  while nums[i]≤0 do
4      l = i+1
5      r = nums.length
6      while l<r do
7          if nums[i]+num[l]+num[r]==0 then
8              print (nums[i],num[l],num[r])
9              while nums[l]==nums[l+1] do l=l+1 // 确保不会重复
10             l = l+1
11             while nums[r]==nums[r-1] do r=r-1 // 确保不会重复
12             r = r-1
13         else if nums[i]+nums[l]+nums[r]<0 then
14             l = l +1
15         else
16             r = r-1
17         end if
18     end while
19     while nums[i]==nums[i+1] do i=i+1 // 确保不会重复
20     i = i+1
21 end while
```

2.3.3. 调试分析

一开始调试遇到的主要问题：即使已经在 9，11 行处确保了 l，r 不会重复，但是仍然会输出重复的三元组。经过分析后发现还需要用同样的方式确保 i，添加了 19 行后便能正确运行。

2.3.4. 总结和体会

该题的难点在于如何在较低的时间复杂度内求出所有的符合要求的三元组。如果使用暴力枚举，则需要 $O(n^3)$ 的时间复杂度。这里我使用了双指针的方法，使得时间复杂度降为 $O(n^2)$。还有一个难点是满足“不重复”的需求，代码中有三处需要进行相应的修改。

2.4. 最大数

问题描述：
给定一组非负整数 nums，重新排列每个数的顺序（每个数不可拆分）使之组成一个最大的整数。

输入要求：
输入包含两行：第一行包含一个整数 n，表示组数 $nums$ 的长度；第二行包含 n 个整数 $nums[i]$。

输出要求：
输出包含一行，为重新排列后得到的数字。

2.4.1. 数据结构设计

```
1 string nums[MAXN]; //利用to_string函数将非负整数转化为string.
```

2.4.2. 功能描述

Function StringCompare(S_1, S_2)

// 判断 S_1 是否排在 S_2 左侧
s1 = S1 + S2
s2 = S2 + S1
for i=1 **to** $s_1.length$ **do**
    if $s_1[i] > s_2[i]$ **then return** *true*
    if $s_1[i] < s_2[i]$ **then return** *false*
end for
return *true*

Function Partition(A,l,r)

// 默认以最后一个元素作为 pivot;l,r 分别为子序列的左右端点下标
// 返回 pivot 的最终下标
i = l - 1
for $j = l$ **to** r-1 **do**
    if *StringCompare(A[j],A[r])* **then** // 使用自定义的比较规则
        i = i + 1
        Swap (A[j],A[i])
    end if
end for
Swap (A[i+1],A[r])
return i+1

Algorithm 2.5: Quick-Sort(A,l,r)

```
1 if l<r then
2 |   p = Partition(A,l,r)
3 |   Quick-Sort (A,l,p-1)
4 |   Quick-Sort (A,p+1,r)
5 end if
```

2.4.3. 调试分析

一开始我自定义的排序规则是：当一个字符串以另一个字符串开头时，哪个小，哪个便排在前面。但是这样的无法正确排序 3，30，34，5，9，会给出 9533034 而非 9534330. 后来我将该比较函数调整为：当一个字符串以另一个字符串开头时，将剩余的字符串再一次与较短的字符串进行比较，以得到最终的结果，但是这样得到的时 9534303. 最后我将自定义比较函数改为上述的逻辑，才成功通过测试。

2.4.4. 总结和体会

本题的基本框架便是排序，唯一需要调整的便是如何判断两个数字串哪个排在前面。而难点便是自定义的比较函数。在本题中我使用的是快速排序，平均时间复杂度为 $\Theta(n \log n)$，这使得在使用了自定的比较函数后，时间复杂度仍能接受。如果使用冒泡排序等简单排序算法，则可能超出时间限制。

2.5. 选取最好的战舰

问题描述：

明天，人类舰队就要迎接三体舰队的探测器——水滴了，作为增援未来部队的你（章北海）刚从“冬眠”中苏醒，就立刻思考起了保留人类文明火种的重大计划，逃亡！为了增加逃亡的成功率，你用 1 秒钟快速了解了所有战舰的历史表现数据（速度、火力）和目前物资储备（食物、燃料），并选出了一艘最合适的战舰。

输入要求：

1. 输入包含若干张表，每一张表表示部分战舰在某个方面的数据表中有若干键值对，键为战舰的名字，值为该战舰的一项数据（均为浮点数）。输入战舰顺序不确定
2. 可以认为每一张表都表示不同的数据（即不会有两张表都表示战舰的加速度），且同一张表中一艘战舰只会出现一次，且每张表中都包含所有战舰的这一个数据（即不会有战舰缺某项数据）
3. 每一项数据都有一个权值，表示你认为这项数据的重要程度。假设表 i 中记录战舰 j 的数据为 tij，表的权值为 wi，那么这艘战舰的总分数计算方法为 cj=t1j*w1+t2j*w2+ ... + tnj*wn
4. 你可以自己假设表格的输入格式细节
5. 要求使用字典树作为基础数据结构

输出要求：

1. 输出一张大表，表的行为战舰名字，列为不同的数据（乘上权值），并打印总分
2. 要求所有战舰按总分从大到小排列，对列的顺序没有要求
3. 你可以自己假设输出格式的细节

2.5.1. 问题分析与解决思路

本题在输入输出方面给了较大的自由度，唯一的限制是使用字典树。这里之所以有必要使用字典树，是因为每张表格的战舰输入顺序不同，导致每输入一个数值，都需要先查找到对应战舰的存储位置，不能根据输入次序直接随机存储，从而不适合使用线性表。

本题不要求在最后的输出中包含表的权重，因此可以在处理表的输入时，便将加权的分数计算出来，后续便不再需要表的权重。

由于字典树的结构无法改动，所以无法直接在它上面进行排序，所以在处理完所有表的输入后，还需要将存储的各个战舰的信息从字典树导出到数组中，以便后续的排序操作。

为了提高效率，用于排序的数组元素中，除了用于排序的 key 之外，便只有指向字典树结点的指针。

2.5.2. 数据结构设计

```
9  string val_name[MAXDATAN];//存储各属性的名称
10 class DT
11 {
```

```
    DT* children[MAXN];//存储指向孩子结点的指针
    prop* data;//存储得分
    char key;//该结点对应的字符
    State state;//是否为某字符串的结尾
    string name;//若该结点为某字符串的结尾，则存储该字符串
};
struct prop
{
    double val[MAXDATAN];//存储各数据的数值
    double sum;//总分数
};
enum State
{
    mid = 0,
    end ,//是某个字符串的结尾
};
struct ElemType
{
    DT* data;//字典树中的一个结点
    double key;//该节点对应的总分数，排序依据
};
struct Array//用于排序
{
    int size;//方便向末端添加元素
    ElemType ships[MAXN];
}
```

2.5.3. 功能描述

Function Print-Ship(ships)

```
打印 val-name// 各属性名称
foreach ship in ships do
    打印 ship →data →name
    打印 ship →data →val
    打印 ship →data →sum
end foreach
```

```
Function Sort-Ship(T)
  Input: 空字典树 T，数据表格 tables
  Output: 按总得分降序排列的数组 ships
1 first = 1
2 foreach table in tables do
3     if first ==1 then
4         first = 0
5         Build-Tree (T,table)
6     else
7         Add-Ship (T,table)
8     end if
9 end foreach
10 ships = Traverse-Tree (T)
11 Quick-Sort (ships)
12 return ships
```

```
Function Build-Tree(T,table)
  Input: 空字典树 T，表格 table
1 foreach table.record do // 遍历每只战舰
2     p = T
3     for i = 1 to record.name.length do
4         j = 1
5         while p→ children[j]≠NULL and p→children[j].key ≠record.name[i] do j+=1
          // 寻找子节点
6         if p → children[j]==NULL then // 未找到
7             新建结点 newNode// state 为 mid,key 为 record.name[i],data 置零
8             p →children[j] = newNode
9         end if
10        p = record.children[j]// 更新指针
11        if i == record.name.length then // 标记为字符串结尾
12            p → state = end
13            更新战舰数据 data
14        end if
15    end for
16 end foreach
```

Function Add-Ship(T,table)

Input: 字典树 T，表格 table

```
1  foreach table.record do // 遍历每只战舰
2      p = T
3      for i = 1 to record.name.length do
4          j = 1
5          while p→ children[j]≠NULL and p→children[j].key ≠record.name[i] do j+=1
6          p = record.children[j]// 更新指针
7          if i = record.name.length then 更新战舰数据 data
8          // 标记为字符串结尾
9      end for
10 end foreach
```

Function Traverse-Tree(T)

Input: 字典树 T

Output: 返回包含字典树中所有状态为 end 的结点的指针数组 ships

```
1  创建空数组 ships
2  i = 1
3  while T →children[i]≠NULL do
4      Traverse (T →children[i],ships)
5      i+=1
6  end while
```

Function Traverse(T,ships)

Input: 字典树结点指针 T，待更新数组 ships

```
1  if T →state == end then
2      向 ships 添加新元素
3  end if
4  i = 1
5  while T →children[i]≠NULL do
6      Traverse (ships,T →children[i])
7      i+=1
8  end while
```

2.5.4. 调试分析

本题在调试过程中并没有遇到太大的问题。排序部分直接使用之前写的代码，字典树部分按照伪代码便能较为轻松地编写出来。

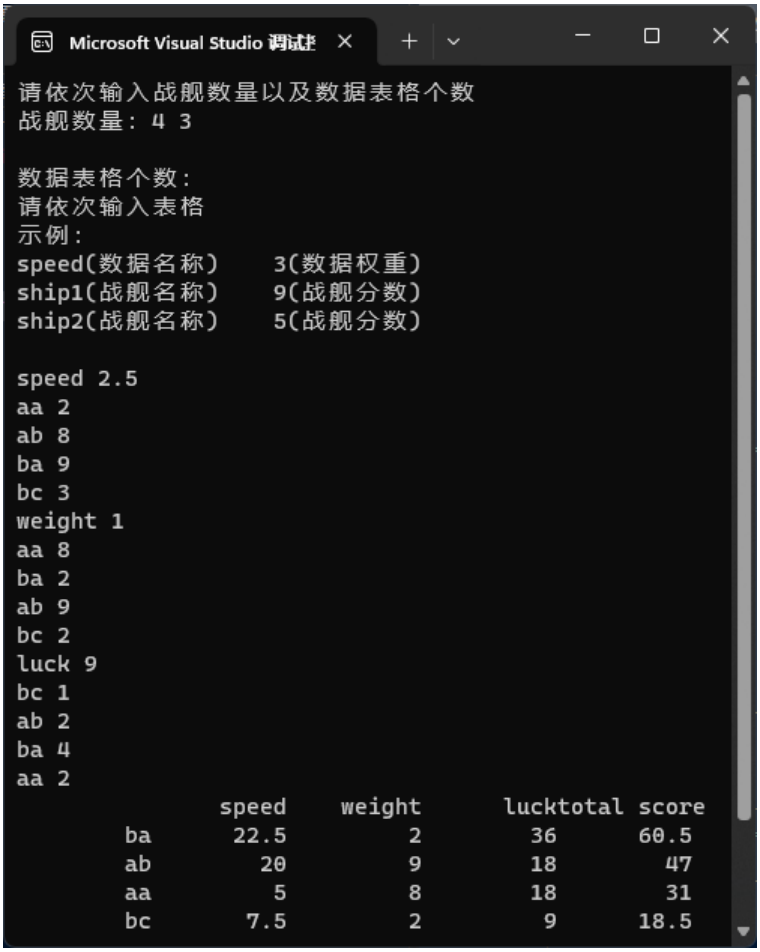

图 2: 测试输入及输出

2.5.5. 总结和体会

数据结构在实际应用中有很大作用，应该根据实际的用途选取适当的数据结构。在本题中，为了能够存储需要被频繁查找的字符串数据，使用了字典树这一数据结构，不仅降低了空间复杂度，也减少了频繁查找所需要的时间。而为了排序，则需要使用线性表加以辅助。从中可以看出灵活使用多种不同的数据结构的重要性。

3. 实验总结

用于排序的各个算法各有特点，即使是高效排序算法，在处理极端和特殊情况时的表现差异也很明显。因此，在实际应用时，需要结合对于输入数据的已有知识，选取最为合适的排序算法。同时，如果只是把排序作为其他算法中的子步骤，在保证时间复杂度的前提下，可以考虑排序算法的稳定排序、部分排序等特性对于算法设计是否有用，从而选取最合适的排序算法。

```
//选取最好的战舰 关键代码
#define MAXDATAN 100
#define MAXN 100
#define INVALID_KEY '@'
string val_name[MAXDATAN];
int cur_val = 0;
int ship_n;
int val_n;
struct prop
{
    double val[MAXDATAN];
    double sum;
};
enum State
{
    mid = 0,
    end,
};
class DT
{
public:
    DT* children[MAXN];
    prop* data;
    char key;
    State state ;
    string name;
    DT() {
        for (int i = 0; i < MAXN; i++)
            children[i] = NULL;
        data = new prop;
        key = INVALID_KEY;
        state = mid;
        name = "";
        data->sum = 0;
    }
};
struct ElemType
{
    DT* data;
    double key;
};
struct Array
{
    int size;
    ElemType ships[MAXN];
};
void build(DT* T)
{
    cin >> val_name[cur_val];
```

```
    double val_weight;
    cin >> val_weight;
    for (int k = 0; k < ship_n; k++) {
        DT* p = T;
        string name;
        cin >> name;
        for (int i = 0; i < name.length(); i++) {
            int j = 0;
            while (p->children[j] != NULL && p->children[j]->key != name[i])
                j++;
            if (p->children[j] == NULL)
                p->children[j] = new DT;
            p = p->children[j];
            p->key = name[i];
            if (i == name.length()-1) {
                p->state = State::end;
                cin >> p->data->val[cur_val];
                p->data->val[cur_val] *= val_weight;
                p->data->sum += p->data->val[cur_val];
                p->name = name;
            }
        }
    }
    cur_val++;
    return;
}
void add(DT* T)
{
    cin >> val_name[cur_val];
    double val_weight;
    cin >> val_weight;
    for (int k = 0; k < ship_n; k++) {
        DT* p = T;
        string name;
        cin >> name;
        for (int i = 0; i < name.length(); i++) {
            int j = 0;
            while (p->children[j] != NULL && p->children[j]->key != name[i])
                j++;
            p = p->children[j];
            if (i == name.length()-1) {
                p->state = State::end;
                cin >> p->data->val[cur_val];
                p->data->val[cur_val] *= val_weight;
                p->data->sum += p->data->val[cur_val];
                p->name = name;
            }
        }
    }
```

```
    cur_val++;
    return;
}
void traverse(DT* T, Array& ships)
{
    if (T->state == State::end) {
        ElemType new_data = { T,T->data->sum };
        ships.ships[ships.size++] = new_data;
    }
    int i = 0;
    while (T->children[i] != NULL) {
        traverse(T->children[i], ships);
        i++;
    }
    return;
}
Array traverse_tree(DT T)
{
    Array ships;
    ships.size = 0;
    int i = 0;
    while (T.children[i] != NULL) {
        traverse(T.children[i], ships);
        i++;
    }
    return ships;
}
```