中山大学计算机学院本科生实验报告

(2025 学年第 1 学期)

课程名称：数据结构与算法实验　　任课教师：张子臻

年级	2024 级	专业（方向）	计算机科学与技术（人工智能与大数据）
学号	242325157	姓名	梁玮麟
电话	18620062182	Email	3371676041@qq.com
开始日期	2025.10.11	结束日期	2025.10.17

第一题

1、实验题目

z5-方程求解

题目描述　已知函数 `y = e^x+ln(x)-1`，实现函数

```
#include "solve.h"
long double solve(long double `y`)
{
// here
}
```

对于传入的 `y`，返回 x 值要求 f(x) 与 `y` 的误差小于 `1e-6`，其中 `0 < y < 1e10`

2、实验目的

求解方程的解，并掌握二分思想，学会给答案预设一个大致区域，然后用二分思想进行搜索。

3、算法设计

设计思路如下：

1. 使用**二分查找**的思想来求解。这也是常见的用来逼近答案的方法。
2. 首先需要找到答案的上下限。因为 `y` 有范围，不难得出 `x` 的**下界**可以是 `0.5`。至于**上界**，可以用一个循环去判断当前上界 `top` 对应的函数值是否比题目给出的 `y` 值大，如果小于 `y` 值就不断地乘二，直至大于等于 `y`。
3. 进入查找环节。首先初始化 `mid=(bottom+top)/2`, 并用一个 **while** 语句进入查找答案的循环，条件设置为 `(exp(top)+log(top)-1-y<0)`：比较当前 `mid` 对应的函数值和 `y` 的差值的绝对值是否大于等于 `1e-6`。如果小于 `1e-6`，说明当前答案已经达到了精度，可以直接返回作为答案；否则继续二分：
   1. 判断 mid 的函数值是否大于 `y`：
      - 如果大于 `y`，说明答案在 [bottom,mid] 之间。令 top=mid。
      - 如果小于 `y`，说明答案在 [mid,top] 之间。令 bottom=mid。
   2. 更新 `mid`，`mid=(bottom+top)/2`.
4. 循环结束后，mid 就是最佳答案，直接输出即可。

细节注意:

- 为了**避免精度不够以及数据过大**的问题出现，使用 `long double`。
- `long double` 对应的**绝对值函数** `fabsl()` 在头文件 `<cmath>` 中。
- 循环中记得**更新 mid** 的值。

具体实现：

```
//z5-方程求解
#include<iostream>
#include<cmath>
#include"solve.h"
using namespace std;
long double solve(long double y){
    long double bottom = 0.5;
    long double top =1;
    while(exp(top)+log(top)-1-y<0){
        top*=2;
    }
    long double mid =(bottom + top)/2;
    while(fabsl(exp(mid)+log(mid)-1-y)>=1e-6){
        if(exp(mid)+log(mid)-1-y>0){
            top =mid;
        }
        else {
            bottom = mid;
        }
        mid = (top+bottom)/2;
    }
    return mid;
}
```

4、程序运行与测试

运行结果：

- 标准输入：

- 实际输出：

```
AC
```

- 期望输出：

```
AC
```

5、实验总结与心得

- 这一题整体主要考察了二分查找的直接应用，难点在于怎么找到**上界**，以及需要注意 long double **类型的绝对值函数**与一般不同的问题。因为输入是 y，所以区间长度大概是 lny，而查找算法本身是 log2(n)(n 为长度)，所以复杂度大概为 $O(\log(\log y))$。

第二题

1、实验题目

z5-Binary Search

题目描述 实现二分查找函数，函数接口如下。

```
#include "binSearch.h"
int binSearch(const int s[], const int size, const int target)
{
```

```
    // 请将实现代码添加在这里
}
```

size 为数组 s 的实际大小。假定 s 非递减有序，如果 s 中存在值为 target 的元素，则返回最后一次出现的位序号，否则返回-1 表示不存在。位序号从 0 开始计。

调用样例

```
int s[8] = {0,1,1,3,3,3,6,6};
cout << binSearch(s,8,3) << endl; //输出 5
cout << binSearch(s,8,4) << endl; //输出-1
```

2、实验目的

使用二分查找，查找对应元素的最后一个出现的位置，并通过这个实验掌握面对不同要求时，**mid 和 bottom,top 的具体关系。**

3、算法设计

设计思路如下：

- 首先是 top 和 bottom 的选择。因为是在给定长度为 size 的数组中查找，所以直接选择让 **bottom=0，top=size-1** 就可以。为了实现找到最后出现的索引的功能，我要让 **top>=target，bottom< target**，使得最终当 top==bottom 时，我们取 result=top，这个 result< target && result>=target，所以这个 result 要么是对应的数，要么不是。而且我们因为取得是 top 而不是 bottom，所以我们的算法**倾向于取相同元素中最后出现的那个位置**，也就实现了我们的目标。
- 对于 mid 的初始化，因为要让 mid 向上取整而不是整除，我们可以让 **mid=(top+bottom+1)/2**，这串式子对于 top+bottom 是奇数或者偶数都适用。
- 平均时间复杂度 O(nlogn)。

1. 为了不断地二分，使用一个 while 循环语句：当 top>bottom 时进入循环：
    1. 比较 mid 对应的数值 s[mid] 与目标数 target 的大小。如果：
        - **s[mid]> target**，说明 target 对应的值应该在 bottom 和 mid 之间。我选择让新的 **top=mid-1。因为，如果 top=mid，此时 s[top] 严格 >target，并非 s[top]>=target。而 top=mid-1，s[mid] 则有机会与 target 取等。**
        - 另一种情况，**s[mid]<=target**，说明 target 对应的值应该在 mid 和 top 之间。我选择让新的 **bottom=mid。此时，s[bottom]。严格小于 target。**
    2. 重新计算 mid。**mid=(top+bottom+1)/2。**
2. 循环结束，直接判断 s[top] 和 target 是否相等。如果相等，返回 top，否则返回-1。

细节注意：

- **必须保证 s[top] 大于等于 target，s[bottom] 严格小于 target。**
- **注意 mid 的计算方式**！若 top+bottom 为奇数，此时需要向上取整；如果是偶数，则无需取整。/是整除，自动向下取整。所以在/2 前先 +1。

具体实现：

```
//z5-Binary Search
#include "binSearch.h"
int binSearch(const int s[], const int size, const int target) {
    int top = size - 1;
    int bottom = 0;
    int mid;
    mid = (top + bottom + 1) / 2;
    while (top > bottom) {
        if (s[mid] > target) {
```

```
            top = mid - 1;
        } else {
            bottom = mid;
        }
        mid = (top + bottom + 1) / 2;
    }
    return s[top] == target ? top : -1;
}
```

4、程序运行与测试

运行结果：

- 标准输入：

- 实际输出：

y

- 期望输出：

y

5、实验总结与心得

- 这一题主要考察的细节就是两个，一个是让 mid 向上取整，另一个是让 top>=target，bottom< target。通过这两个设计，再加上二分查找，就可以找到对应元素的最大索引。

第三题

1、实验题目

z5-最大值最小化

题目描述 把一个包含 n 个正整数的序列划分成 m 个连续的子序列（每个正整数恰好属于一个序列）。设第 i 个序列的各数之和为 `S(i)`，如何让所有 `S(i)` 的最大值尽量小？例如序列 `1 2 3 2 5 4`，划分成 3 个序列的最优方案为 `1 2 3 | 2 5 | 4`，其中 `S(1)=6`，`S(2)=7`，`S(3)=4`，最大值为 `7`；如果划分成 `1 2 | 3 2 | 5 4`，则最大值为 `9`，不如刚才的好。$n \leq 10^6$，所有数之和不超过 10^9。

输入描述 可能有多个输入样例。每个样例第一行输入两个整数 n 和 m，第二行输入 n 个整数。

输出描述 输出所有子序列划分中子序列和的最大值的最小值。

输入样例

```
6 3
1 2 3 2 5 4
```

输出样例

```
7
```

2、实验目的

通过完成实验，进一步加深对二分思想的理解，学会将二分思想迁移到查找问题之外。

3、算法设计

设计思路如下：

- 为了找到最大值的最小值，我们可以假设有**一个答案集合**，只要在这个集合中找到正确答案就可以。找正确答案的过程可以用**二分查找**，然后对于某个元素，只要**验证能否完成划分即可。**
- 时间复杂度 O(nlogn)。

1. 记录各个输入。
2. 答案的最小值是这一串数中的最大值，答案的最大值是这一些数的总和。所以我们要求出这些数的和 sum，并维护 max，存储这些数的最大值。让 **bottom = max, top = sum，取出第一个数作为可能的答案 x=(top+bottom)/2。**
3. 用一个 while(top>bottom) 语句进行二分查找：:
    1. 我用 sum 以及 times 分别表示子序列中的总和以及整个序列中可以画的 | 的次数。**其中 times 初始化为子序列数-1。**
    2. for 循环，用 j 遍历序列。如果 j < n && times >= 0，则继续遍历：
        - 如果 sum 加上 j 指向的数依然小于等于 x，则可以加上这个数，且 j++。
        - 如果 sum 加上 j 指向的数之后大于 x，那么就要在这里画 |，同时 times--，让 sum 重置为 0。
    3. 判断 **times 是否大于等于零。**如果是，说明这个 x 就是理想情况或者比理想情况大，将这个 x 设置为 top；否则就是 times 小于零，| 的使用次数超标，x 太小了，将 bottom 设置为 x+1。**(这里 bottom<x<=top)**
    4. 更新 x=(top+bottom)/2。
4. 最后跳出了循环，说明 top==bottom，此时 top 对应的就是正确答案 (**因为 top 可以和 x 取等**)。

细节注意：

- 需要开 long long！否则数据范围不够！
- 有多个样例输入，需要使用 while(cin>>n) 判断是否还有输入。

具体实现：

```
//z5-最大值最小化
#include <iostream>
#include <cmath>
#include <vector>
using namespace std;
typedef long long ll;
int main() {
    ll n, m;
    while(cin>>n) {
        cin >> m;
        vector<ll> num;
        ll sum = 0;
        ll max = -1;
        for (ll i = 0; i < n; i++) {
            ll a;
            cin >> a;
            sum += a;
            num.push_back(a);
            if(a>=max) max=a;
        }
        ll bottom = max, top = sum;
        ll x = (bottom + top) / 2;
        while (bottom < top) {
            // 求得区间最大值
```

```
            ll sum = 0;
            ll times = m - 1; // 可以划几根棒子分割
            for (ll j = 0; j < n && times >= 0;) {
                if (sum + num[j] <= x) {
                    sum += num[j];
                    j++;
                } else {
                    sum = 0;
                    times--;
                }
            }
            if (times >= 0) { // 当前 x 太大
                top = x;
            } else {
                bottom = x + 1;
            }
            x = (bottom + top) / 2;
        }
        cout << top << endl;
    }
    return 0;
}
```

4、程序运行与测试

运行结果：

数据点 0

- 标准输入：

```
6 3
1 2 3 2 5 4
```

- 实际输出：

```
7
```

- 期望输出：

```
7
```

数据点 1

- 标准输入：

```
14241 2633
1897 12278 6388 11961 20771 31975 17924 13605 14667 25962 28612 15171 5388 7954 5712 7767 29680 17741 18
```

- 实际输出：

```
99594
32756
38857
419321
36677
155142
35290
69490
```

```
43835
553618
```

- 期望输出：

```
99594
32756
38857
419321
36677
155142
35290
69490
43835
553618
```

5、实验总结与心得

- 这一题很有趣，需要先找到答案的所有可能情况，然后用二分查找的思路去做。我在一开始想的是，用平均数去近似为最好结果，然后再让每个子序列尽可能按照平均数的分分配方法去分。后面发现并不合理。所以，其实二分查找的思想并不局限在查找，需要我们遇到问题灵活变通。