

华中科技大学

课程实验报告

课程名称： C 语言程序设计实验

专业班级： 计算机科学与技术学院 1904 班

学 号： U201914981

姓 名： 王 翰 辉

指导教师： 王 多 强

报告日期： 2019-12-31

计算机科学与技术学院

目 录

1 流程控制实验	1
1.1 程序改错	1
1.2 程序修改替换	3
1.3 程序设计	6
1.4 小结	26
2 数组程序设计实验	28
2.1 程序改错与跟踪调试	28
2.2 程序完善与修改替换	31
2.3 程序设计	38
2.4 小结	59
3 结构与联合实验	60
3.1 表达式求值的程序验证	60
3.2 源程序修改替换	62
3.3 程序设计	67
3.4 小结	106
参考文献	108

1 流程控制实验

1.1 程序改错

/*原版程序*/

```
1. #include <stdio.h>
2. int main( )
3. {
4. int i, x, k, flag = 0;
5. printf("input a intergrate bigger than 1, end with Ctrl+z\n");
6. while(scanf("%d", &x) != EOF)
7. {
8. for(i = 2, k = x>>1; i <= k; i++)
9. if(!x % i)
10. {
11. flag = 1;
12. break;
13. }
14. if(flag = 1)
15. printf("%d is a composit number\n", x);
16. else
17. printf("%d is not a composit number\n", x);
18. }
19. return 0;
20. }
```

错误分析即修改：（单独分析某处时假定其余位置已经修改完善）

1. flag = 0

分析一若只在这里初始化 `flag = 0`，按照程序逻辑，当某一次输入了一个合数之后，`flag` 的值就被改变为 1，往后没有再变回 0，所以之后的输入不论什么数都会被判定为合数。

修改一应当在每次循环中加入对 `flag` 赋值为 0，如可在 7 行之后加入 `flag = 0`，以此保证每判断一个新的数字时 `flag` 初值为 0。

2. `!x % i`

分析—逻辑非运算符的优先级高于取模运算符，故这里先将 `x` 取非，若输入非 0，则 `!x == 0`，对 `i` 取模恒为 0，故判断恒为质数。

修改一应改为 `!(x % i)`

3. `flag = 1`

分析—这里并没有判断 `flag` 是否为 1，而是把 1 赋给了 `flag`。

修改一应改为 `flag == 1`

`/*修改*/`

```
1. #include <stdio.h>
2. int main()
3. {
4.     int i, x, k, flag;
5.     printf("input a intergrate bigger than 1, end with Ctrl+z\n");
6.     while(scanf("%d", &x) != EOF)
7.     {
8.         flag = 0;
9.         for(i = 2, k = x >> 1; i <= k; i++)
10.            if(!(x % i))
11.            {
12.                flag = 1;
13.                break;
14.            }
15.            if(flag == 1)
16.                printf("%d is a composit number\n", x);
17.            else
18.                printf("%d is not a composit number\n", x);
19.        }
20.    return 0;
```

```

C:\WINDOWS\system32\cmd.exe
input a intergrate bigger than 1, end with Ctrl+z
2
2 is not a composit number
4
4 is a composit number
6
6 is a composit number
7
7 is not a composit number
9
9 is a composit number
29
29 is not a composit number
z
请按任意键继续. . .

```

图 1.1-1 原版修正

1.2 程序修改替换

(1) 改为单出口

原版中的 `for` 循环有两出口即两种退出循环的方式，一种是运行到最后，不符合 `for` 中的循环条件而退出，另一种则是被判定为合数，触发了 `break` 机制进而退出了循环。如果不允许使用 `break` 等控制语句有两类方法。1) 直接去掉 `break` 语句，如此符合了单出口的要求，但是这么做会进行无谓的运算，降低了代码执行效率。2) 将触发 `break` 的条件加入到循环条件中。即 `for` 中加入要求：flag 为 0 循环才能继续，否则跳出。这里我们采用方案 2)。

/*单出口*/

```

1. #include <stdio.h>
2. int main()
3. {
4.     int i, x, k, flag;
5.     printf("input a intergrate bigger than 1, end with Ctrl+z\n");
6.     while(scanf("%d", &x) != EOF)
7.     {
8.         flag = 0;
9.         for(i = 2, k = x >> 1; i <= k && !flag; i++)
10.            if(!(x % i))
11.                flag = 1;
12.         if(flag == 1)

```

```

13.     printf("%d is a composit number\n", x);
14.     else
15.     printf("%d is not a composit number\n", x);
16. }
17. return 0;
18.}

```

```

C:\WINDOWS\system32\cmd.exe
input a intergrate bigger than 1, end with Ctrl+z
34534421
34534421 is a composit number
2
2 is not a composit number
3331
3331 is not a composit number
24
24 is a composit number
请按任意键继续. . .

```

图 1.2-1 单出口版

(2) do-while 循环版

原版中采用了 for 循环，这样就能够避免在输入值为 2 时进入循环（一开始就不符合循环条件）。但是如果改用 do-while 就无论如何都会执行一次循环，若不做逻辑修改就会把 2 也当作合数。

/*do-while 版*/

```

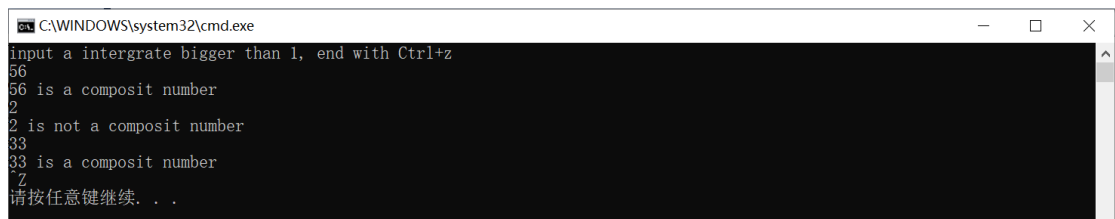
1. #include <stdio.h>
2. int main()
3. {
4.     int i, x, k, flag;
5.     printf("input a intergrate bigger than 1, end with Ctrl+z\n");
6.     while(scanf("%d", &x) != EOF)
7.     {
8.         i = 2;
9.         flag = 0;
10.        do{
11.            if(!(x % i))
12.                flag = 1;
13.            i++;
14.        }while(i < (k = x>>1) && !flag);

```

```

15.     if(x != 2 && flag == 1)
16.         printf("%d is a composit number\n", x);
17.     else
18.         printf("%d is not a composit number\n", x);
19. }
20. return 0;
21.}

```



```

C:\WINDOWS\system32\cmd.exe
input a intergrate bigger than 1, end with Ctrl+z
56
56 is a composit number
2
2 is not a composit number
33
33 is a composit number
7
7 is a composit number
请按任意键继续. . .

```

图 1.2-2 do-while 循环版

(3) 三位纯粹合数

首先注意到纯粹合数的首位只有可能为 4, 6, 8, 9, 因此对于首位不必从 1 到 9 全部测试。然后我们只需仿照上述判断合数的方法对三位数, 两位数进行盘算即可。

/*三位纯粹合数*/

```

1. #include <stdio.h>
2. int main()
3. {
4.     int a[] = {4, 6, 8, 9}, t1, t2, k, k0;
5.     for(int i = 0; i < 4; i++)
6.         for(int j = 0; j < 100; j++)
7.             {
8.                 k = a[i] * 100 + j;    //符合要求的三位数百位必为 4 或 6 或 8 或
9.                 9
9.                 t1 = t2 = 0;
10.                for(int p = 2; p < (k>>1) && !t1; p++)
11.                    if(!(k % p))
12.                        t1 = 1;

```

```

13.     if(t1)
14.     {
15.         k0 = k / 10;
16.         for(int p = 2; p < (k0>>1) && !t2; p++)
17.             if(!(k0 % p))
18.                 t2 = 1;
19.     }
20.     if(t2)
21.         printf("%d ", k);
22.     }
23.     printf("\n");
24.     return 0;
25. }

```

The screenshot shows a Windows command prompt window with the title bar 'C:\WINDOWS\system32\cmd.exe'. The window contains a list of numbers from 400 to 998, arranged in rows of 10. The prompt '请按任意键继续...' is visible at the bottom of the window.

图 1.2-3 纯粹合数

1.3 程序设计

(1)

本题要求使用 if-else 语句和 switch 语句分别给出对应程序。这里第一种给出伪码算法步骤：

- 1) 定义整型变量 x，双精度浮点型变量 y = 0
- 2) 输入 x
- 3) if 1000<=x<2000, x 超出 1000 部分按 5%计税赋给 y，执行 8)

- 4) else if $x < 3000$, x 超出 2000 部分按 10% 计税再加上 50 赋给 y , 执行 8)
- 5) else if $x < 4000$, x 超出 3000 部分按 15% 计税再加上 150 赋给 y , 执行 8)
- 6) else if $x < 5000$, x 超出 4000 部分按 20% 计税再加上 300 赋给 y , 执行 8)
- 7) else x 超出 5000 部分按 25% 计税再加上 500 赋给 y , 执行 8)
- 8) 输出 y

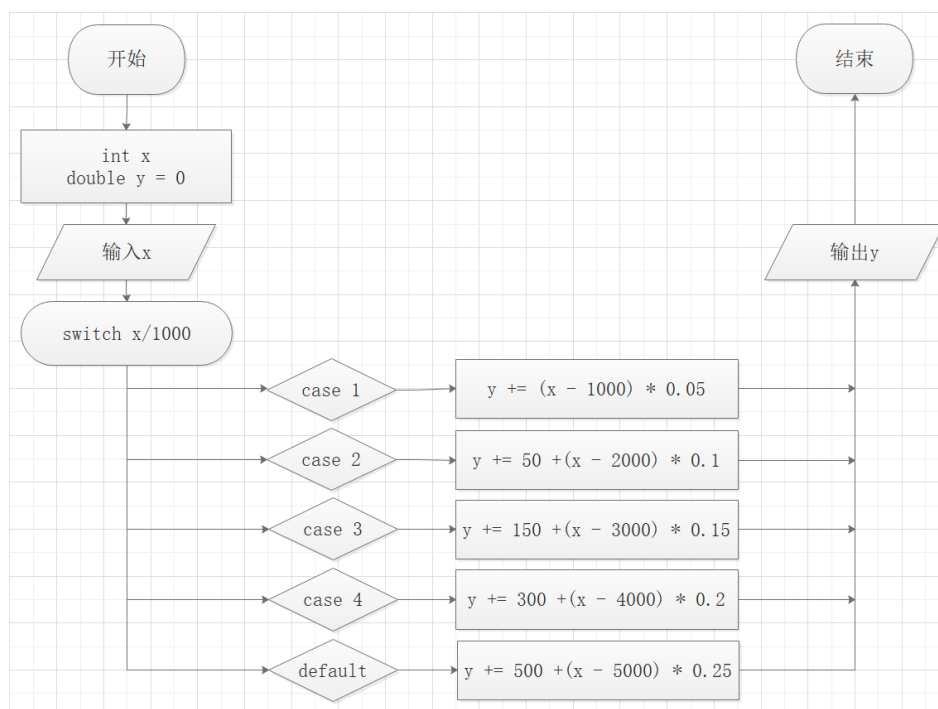
```
1. #include <stdio.h>
2. int main()
3. {
4.     int x;
5.     double y = 0;
6.     scanf("%d", &x);
7.     if(x < 2000)
8.         y += (x - 1000) * 0.05;
9.     else if(x < 3000)
10.        y += 50 + (x - 2000) * 0.1;
11.    else if(x < 4000)
12.        y += 150 + (x - 3000) * 0.15;
13.    else if(x < 5000)
14.        y += 300 + (x - 4000) * 0.2;
15.    else
16.        y += 500 + (x - 5000) * 0.25;
17.    printf("%f\n", y);
18.    return 0;
19.}
```



```
C:\WINDOWS\system32\cmd.exe
3245
186.750000
请按任意键继续. . .
```

图 1.3-1.1 if-else

第二种给出流程图:



```
1. #include <stdio.h>
```

```
2. int main()
```

```
3. {
```

```
4.     int x;
```

```
5.     double y = 0;
```

```
6.     scanf("%d", &x);
```

```
7.     switch(x / 1000)
```

```
8.     {
```

```
9.         case 1:y += (x - 1000) * 0.05;break;
```

```
10.        case 2:y += 50 + (x - 2000) * 0.1;break;
```

```
11.        case 3:y += 150 + (x - 3000) * 0.15;break;
```

```
12.        case 4:y += 300 + (x - 4000) * 0.2;break;
```

```
13.        default:y += 500 + (x - 5000) * 0.25;
```

```
14.    }
```

```
15.    printf("%f\n", y);
```

```
16.    return 0;
```

```
17.}
```

C:\WINDOWS\system32\cmd.exe

6758

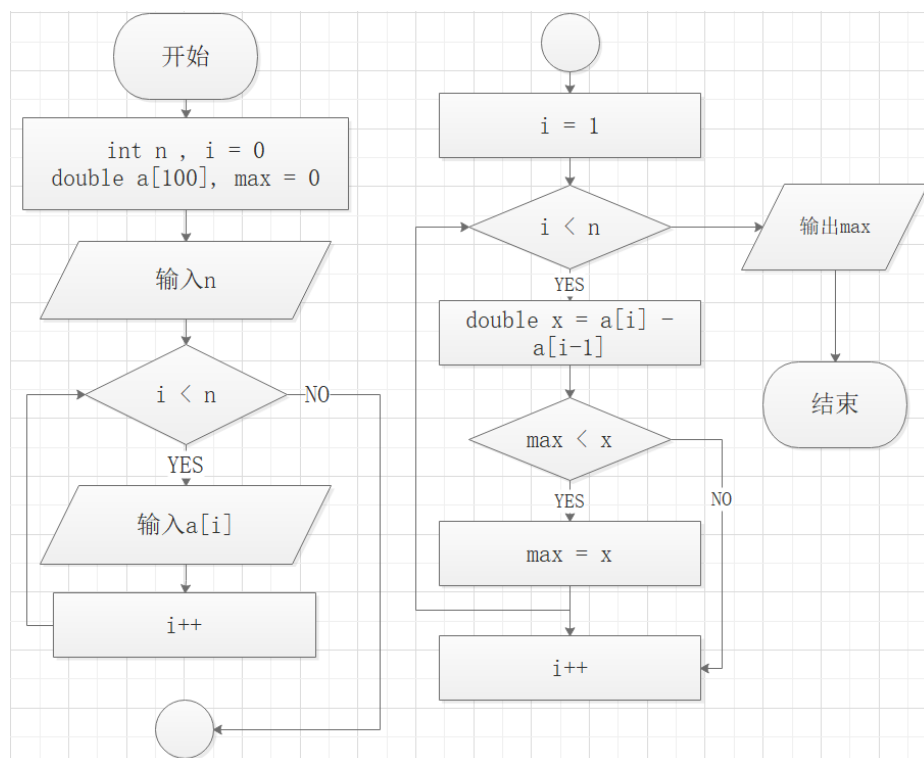
939.500000

请按任意键继续. . .

图 1.3-1.2 switch-case

(2)

这一题的实现可以采用两类思路，一种将所有数据都保留下来，存储在数组内，完全输入完后再进行判断，另一种则可以输入后立刻进行判断，这里我们采用第一种思路。



```
1. #include <stdio.h>
2. #include <math.h>
3. int main()
4. {
5.     double a[100];
6.     int n;
7.     scanf("%d", &n);
8.     for(int i = 0; i < n; i++)
9.         scanf("%lf", &a[i]);
10.    double max = 0;
11.    for(int i = 1; i < n; i++)
```

```

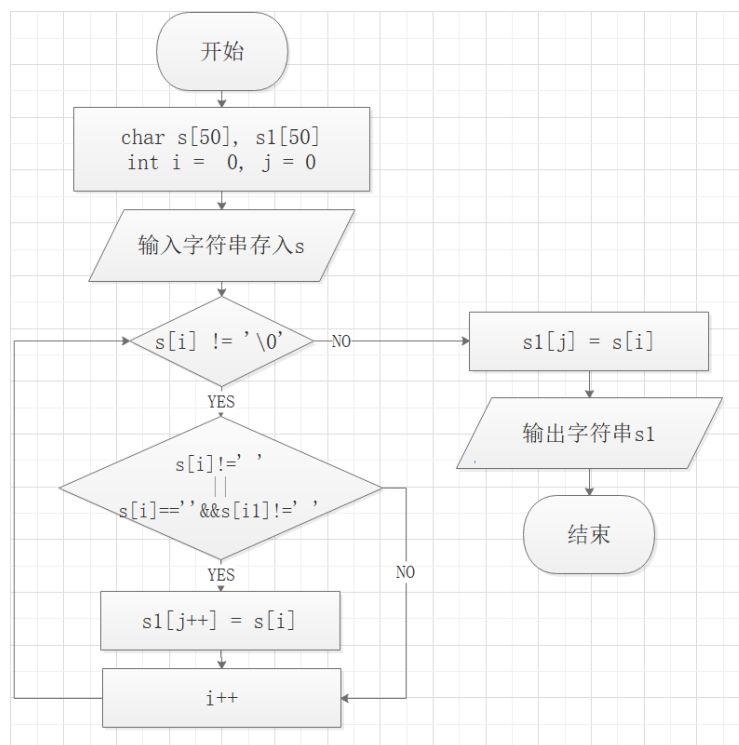
12.    max = (max < fabs(a[i] - a[i-1]))?fabs(a[i] - a[i-1]):max;
13.    printf("%f\n", max);
14.    return 0;
15.}

```

图 1.3-2 最大波动

(3)

这一题的关键在于判断当前字符是否为空格，之前的字符串中是否有连续的空格。同样可以将字符串存储起来再判断或者不存储直接判断。由于这里只给出第一种流程图表示，第二种则直接给出代码及简单分析。



```

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5.     char s[50], s1[50];
6.     fgets(s,sizeof(s),stdin);

```

```

7.  int i, j;
8.  for(i = 0, j = 0; s[i] != 0; i++)
9.  {
10.     if((s[i] == ' ' && s[i + 1] != ' ') || s[i] != ' ')
11.         s1[j++] = s[i];
12. }
13. s1[j] = s[i];
14. printf("%s", s1);
15.}

```

表 1-1 编程题 3-3 的测试数据

测试用例	程 序 输 入	理 论 结 果
用例一	(四个空格) asdf asdf sdfdsf fs fdg	(一个空格) asdf asdf sdfdsf fdg
用例二	qwert qwert qwert qwert	qwert qwert qwert qwert
用例三	(五个空格) qwert qwert qwert	(一个空格) qwert qwert qwert

```

C:\WINDOWS\system32\cmd.exe
asdf asdf sdfdsf fs fdg
asdf asdf sdfdsf fs fdg
请按任意键继续. . .

```

图 1.3-3.1 fgets()版

第二种方法我们在这里使用了我们所学的状态表示。定义 `state == 0` 为其余字符状态，`state == 1` 为空格字符状态。进入某一字符状态表明该字符的上一个字符（该字符为第一个字符且为空格要单独讨论）。若 `state == 1` 且当前字符也为空格则不输出，否则输出当前字符并赋值 `state = 0`。若 `state == 0` 且当前字符为空格则输出空格并赋值 `state = 1`，否则只输出当前字符。

```
1. #include <stdio.h>
2. int main()
3. {
4.     char c;
5.     int state;
6.     c = getchar();
7.     if(c == ' ')
8.     {
9.         state = 1;
10.        putchar(c);
11.    }
12.    else
13.        state = 0;
14.    while(c != EOF)
15.    {
16.        switch(state)
17.        {
18.            case 0:
19.                putchar(c);
20.                if(c == ' ')
21.                    state = 1;
22.                break;
23.            case 1:
24.                if(c != ' ')
25.                {
26.                    putchar(c);
27.                    state = 0;
28.                }
29.                break;
```

```
30.     }
31.     c = getchar();
32. }
33. return 0;
34. }
```


C:\WINDOWS\system32\cmd.exe

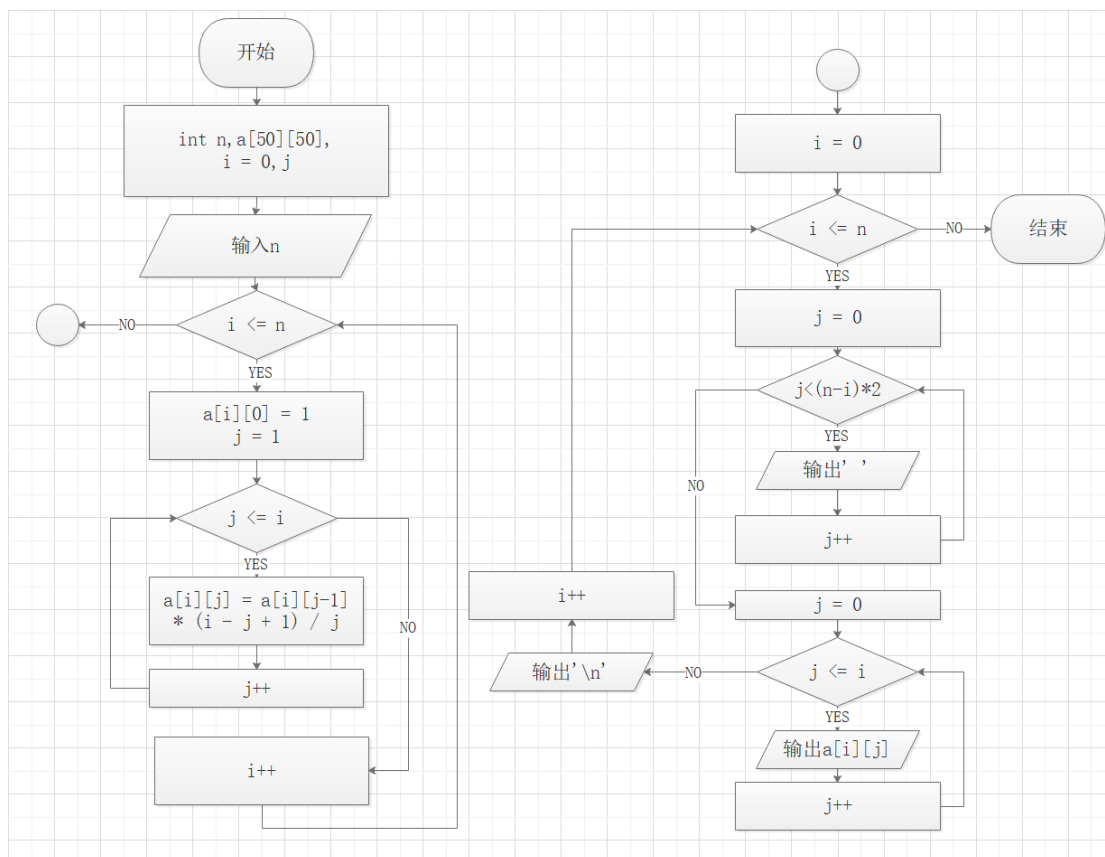
```

qwerty qwerty  qwerty  qwerty
qwerty qwerty qwerty qwerty
      qwerty qwerty      qwerty
qwerty qwerty qwerty
^Z
请按任意键继续. . .
```

1.3-3.2 getchar()-putchar()版

(4)

这一题的要求是要将杨辉三角以类似正三角形式输出，而非一般的下三角形式。而对应的空格数目要求则可以通过设置域宽为 4 完成。流程图的左半完成了 n 阶杨辉三角的计算与储存，右半则完成了空格与数据的输出。



```
1. #include <stdio.h>
2. int main()
3. {
4.     int a[50][50];
5.     int n;
6.     scanf("%d", &n);
7.     for(int i = 0; i <= n; i++)
8.     {
9.         a[i][0] = 1;
10.        for(int j = 1; j <= i; j++)
11.            a[i][j] = a[i][j-1] * (i - j + 1) / j;
12.    }
13.    for(int i = 0; i <= n; i++)
14.    {
15.        for(int j = 0; j < (n - i) * 2; j++)
16.            printf(" ");
17.        for(int j = 0; j <= i; j++)
18.        {
19.            if(j == 0)
20.                printf("%d", a[i][j]);
21.            else
22.                printf("%4d", a[i][j]); //条件的要求等同于域宽为 4
23.        }
24.        printf("\n");
25.    }
26.    return 0;
27.}
```

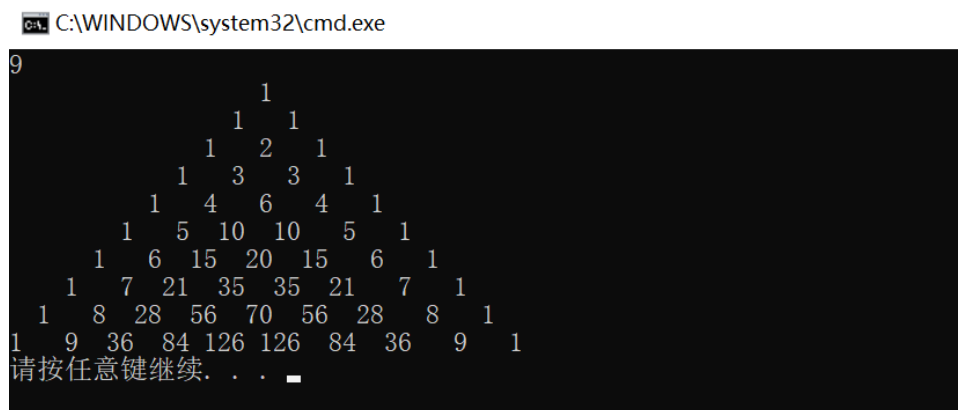



图 1.3-4 杨辉三角

(5)

这类题目可以说让你自己去做会采用最为简单的方法，比如说易知该四位数必为 121 的倍数，那么就可以用几个完全平方数去乘 121 就可以很快锁定答案 7744。但是如果把这一思路直接变为程序让计算机去执行并没有体现出计算机对简单枚举类问题的快速计算的优越性，因此我们在这里使用一种显得比较“笨”的方法。将所有可能的完全平方数列举出来，分解为 4 个数字，判断千位是否与百位相等，十位与个位是否相等，十位与百位是否相等从而判断是否符合条件，将符合条件的数输出。

(由于空间缘故，先给出代码)

```
1. #include <stdio.h>
2. int main()
3. {
4.     for(int i = 32; i <= 99; i++)
5.     {
6.         int a = i * i;
7.         int b[4];
8.         for(int j = 0; j < 4; j++)
9.         {
10.            b[j] = a % 10;
11.            a /= 10;
12.        }
```

```

13.     if(b[0] == b[1] && b[2] == b[3] && b[0] != b[2])
14.         printf("%d\n", i * i);
15.     }
16.     return 0;
17. }

```

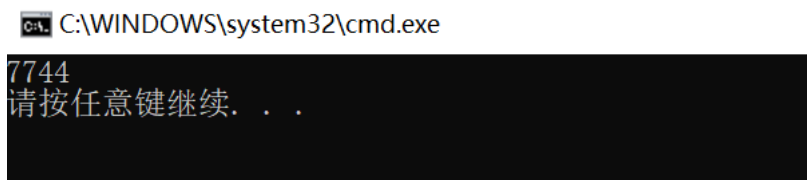
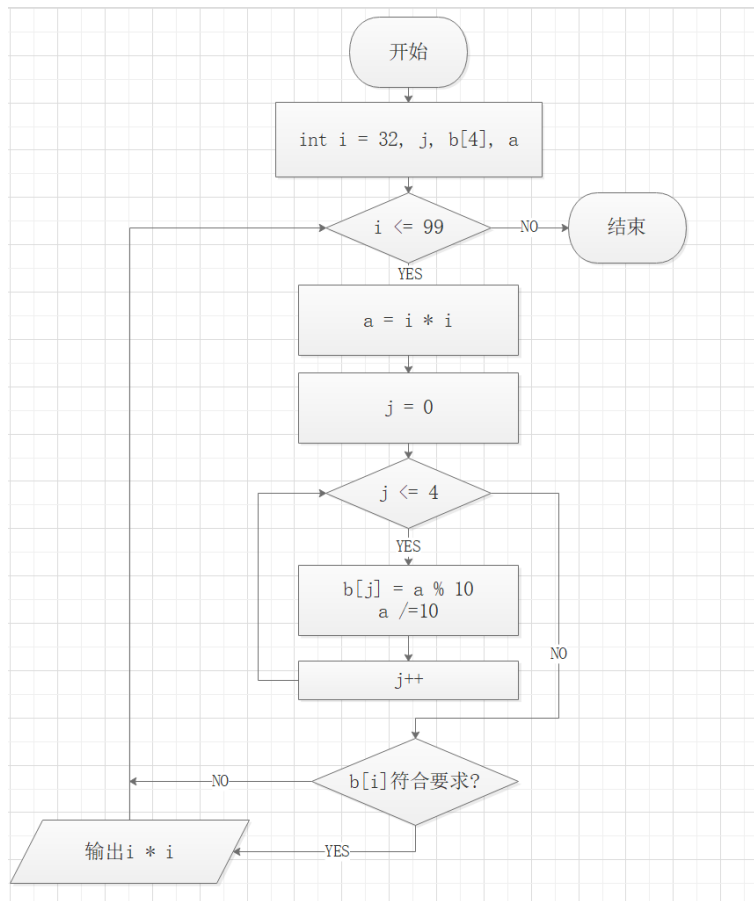


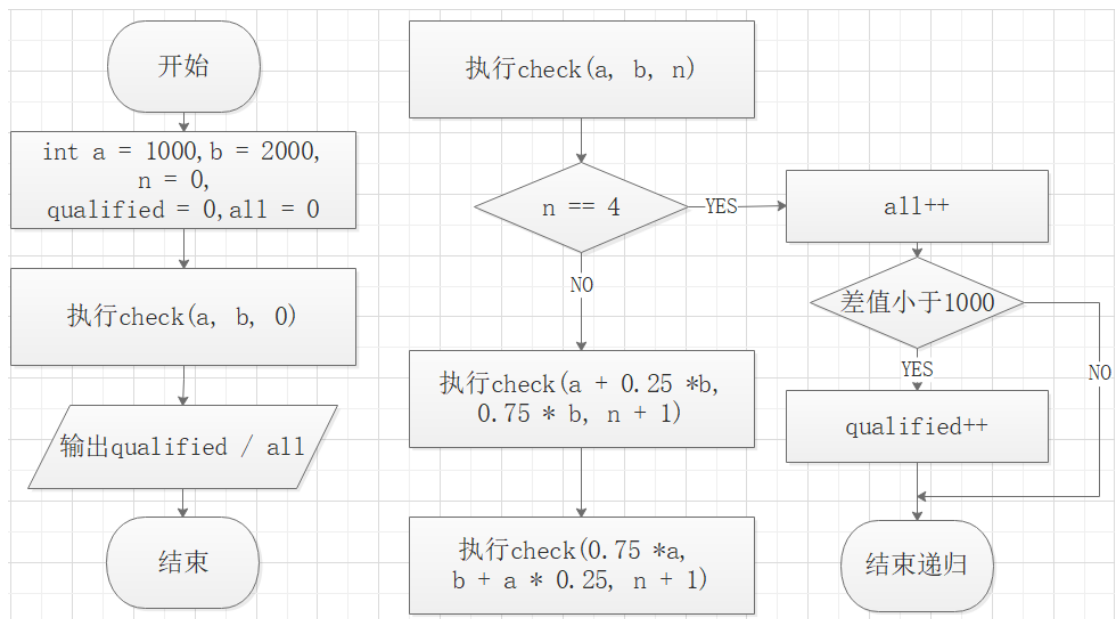
图 1.3-5 确定车牌

(6)

这一题是要求解概率，很容易得到两种思路，一种是用频率近似代替概率，我们进行充分多次的实验，然后计算符合要求的次数和总数之比，这一种方法就

可以充分地运用我们所学的生成一定范围内的伪随机数。例如生成 1, 2, 其中为 1 时甲胜, 为 2 时乙胜, 执行四次再不断进行上述实验即可。另一种则是精确地枚举每一种可能, 直接计算出概率。这里我采用递归的方法去实现第二种思路。

(可能对于如何绘制递归算法的流程图有些不清楚导致绘图不规范, 但是意思应当较为清楚)



```

1. #include <stdio.h>
2. #include <math.h>
3. int qualified;
4. void check(double a, double b, int n0, int n);
5. int main()
6. {
7.     int n, all = 1;
8.     double a, b;
9.     scanf("%lf%lf%d", &a, &b, &n);
10.    for(int i = 0; i < n; i++)
11.        all *= 2;
12.    check(a, b, 0, n);
13.    printf("qualified rate: %f\n", (double)qualified / all);

```

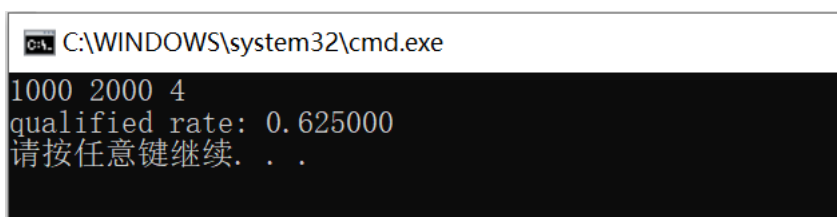
```

14. return 0;
15.}
16.
17. void check(double a, double b, int n0, int n)
18.{
19.    if(n0 == n)
20.    {
21.        if(fabs(a - b) < 1000)
22.            qualified++;
23.    }
24.    else
25.    {
26.        double a0, b0;
27.        a0 = a + 0.25 * b, b0 = 0.75 * b;
28.        check(a0, b0, n0 + 1, n);
29.        a0 = 0.75 * a, b0 = b + 0.25 * a;
30.        check(a0, b0, n0 + 1, n);
31.    }
32.}

```

注：

流程图只是给出了求解 $a == 1000$ ， $b == 2000$ ， $round == 4$ 的这一组特殊数据的解法，而我实际编写的程序则是采取了输入 a ， b 以及 $n0$ （代表 $round$ 轮数），并且通过直接计算 2^{n0} 来表示 all （总次数）。如此就可以多进行数据更改后的实验。



```

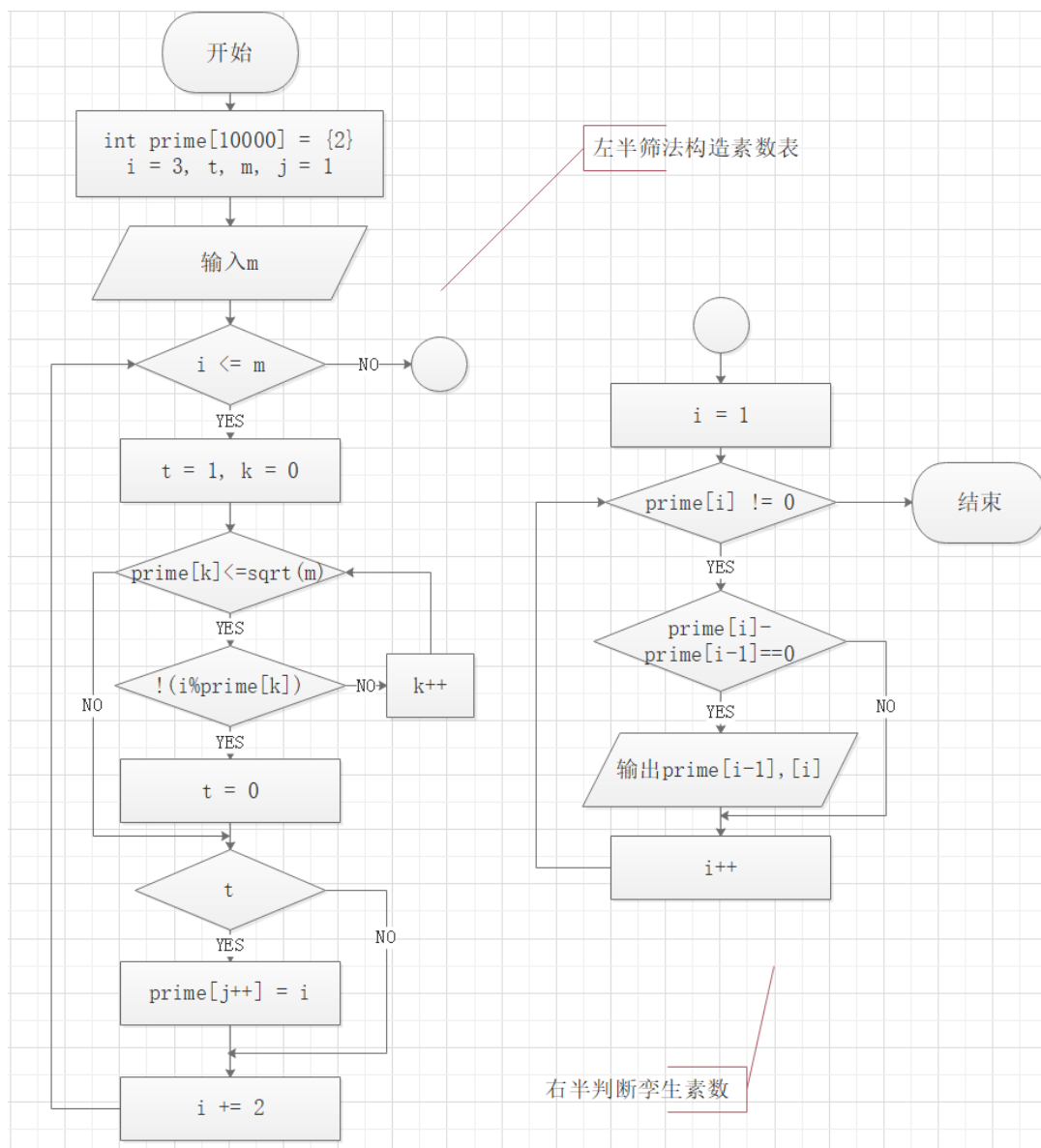
C:\WINDOWS\system32\cmd.exe
1000 2000 4
qualified rate: 0.625000
请按任意键继续. . .

```

图 1.3-6 对战

(7)

本题核心是通过筛法通过已知的素数来构建一张足够大的素数表，然后再在这张表里查找孪生素数。所以代码和流程图的绘制当中也体现了将两大主体部分主体部分分离展现的方式。



1. #include <stdio.h>
2. #include <math.h>
3. #include <string.h>

```

4.  int main()
5.  {
6.      int m, i, prime[10000], t;
7.      memset(prime,0,sizeof(prime));
8.      prime[0] = 2;
9.      //以上两行可在初始化阶段完成
10.     scanf("%d", &m);
11.     for(int j = 1, i = 3; i <= m; i += 2)
12.     {
13.         t = 1;
14.         for(int k = 0; prime[k] <= sqrt((double)i); k++)
15.             if(i % prime[k] == 0)
16.             {
17.                 t = 0;
18.                 break;
19.             }
20.         if(t)
21.             prime[j++] = i;
22.     }
23.     //完成素数表的构建
24.     t = 0;
25.     for(i = 1; prime[i] != 0; i++)
26.     {
27.         if(prime[i] - prime[i-1] == 2)
28.         {
29.             printf("(%d,%d)\t\t", prime[i-1], prime[i]);
30.             t++;
31.         }
32.         //判断是否为孪生素数
33.         if(t == 5)

```

```

34.  {
35.    printf("\n");
36.    t = 0;
37.  }
38.
39. }
40. if(t)
41.    printf("\n");
42. return 0;
43. }

```

```

C:\WINDOWS\system32\cmd.exe
1000
(3, 5)      (5, 7)      (11, 13)     (17, 19)     (29, 31)
(41, 43)    (59, 61)    (71, 73)    (101, 103)   (107, 109)
(137, 139)  (149, 151)  (179, 181)  (191, 193)   (197, 199)
(227, 229)  (239, 241)  (269, 271)  (281, 283)   (311, 313)
(347, 349)  (419, 421)  (431, 433)  (461, 463)   (521, 523)
(569, 571)  (599, 601)  (617, 619)  (641, 643)   (659, 661)
(809, 811)  (821, 823)  (827, 829)  (857, 859)   (881, 883)
请按任意键继续. . .

```

图 1.3-7 孪生素数

(8)

这一题要求两个“五位数”每一位的数字都不相同，直观的想法可以按照书本例题中的枚举法进行十次迭代以保证数字不同。但是这样重复的代码过多，这是我们自然会想到利用函数递归来对程序的代码进行精简。只用把当前元素数值是否与之前的元素数值相同的判断编写一次即可。

注：在此处我加入了“clock”在最后一显示秒数查看两者耗时比较。

迭代版：

```

1. #include <stdio.h>
2. #include <time.h>
3. clock_t start, stop;

```

```

4.  int main()
5.  {
6.      int x;
7.      scanf("%d", &x);
8.      start = clock();
9.      for(int a = 1; a <= 9; a++)
10.     {
11.         for(int b = 0; b <= 9; b++)
12.         {
13.             if(b == a)
14.                 continue;
15.             for(int c = 0; c <= 9; c++)
16.             {
17.                 if((c-b)*(c-a) == 0)
18.                     continue;
19.                 for(int d = 0; d <= 9; d++)
20.                 {
21.                     if((d-c)*(d-b)*(d-a) == 0)
22.                         continue;
23.                     for(int e = 0; e <= 9; e++)
24.                     {
25.                         if((e-d)*(e-c)*(e-b)*(e-a) == 0)
26.                             continue;
27.                         for(int f = 0; f < a; f++)
28.                         {
29.                             if((f-e)*(f-d)*(f-c)*(f-b) == 0)
30.                                 continue;
31.                             for(int g = 0; g <= 9; g++)
32.                             {
33.                                 if((g-f)*(g-e)*(g-d)*(g-c)*(g-b)*(g-a) == 0)

```



```

34.         continue;
35.     for(int h = 0; h <= 9; h++)
36.     {
37.         if((h-g)*(h-f)*(h-e)*(h-d)*(h-c)*(h-b)*(h-a) == 0)
38.             continue;
39.         for(int i = 0; i <= 9; i++)
40.         {
41.             if((i-h)*(i-g)*(i-f)*(i-e)*(i-d)*(i-c)*(i-b)*(i-a) == 0)
42.                 continue;
43.             for(int j = 0; j <= 9; j++)
44.             {
45.                 if((j-i)*(j-h)*(j-g)*(j-f)*(j-e)*(j-d)*(j-c)*(j-b)*(j-
a) == 0)
46.                     continue;
47.                 if(x * (f*10000+g*1000+h*100+i*10+j) == a*10000
+b*1000+c*100+d*10+e)
48.                     printf("%d%d%d%d%d/%d%d%d%d%d=%d\n
", a, b, c, d, e, f, g, h, i, j, x);
49.             }
50.         }
51.     }
52. }
53. }
54. }
55. }
56. }
57. }
58. }
59. stop = clock();
60. printf("%f", (double)(stop - start)/CLK_TCK);

```

61. **return** 0;

62. }

63. //在这里枚举算法的代码就显得较为复杂了,从 a 到 j 代码本质没有改变,
只是输入复杂度的提高,可以考虑用函数代替。

C:\WINDOWS\system32\cmd.exe

```
35
48265/01379=35
63945/01827=35
64295/01837=35
74865/02139=35
93485/02671=35
0.302000请按任意键继续. . .
```

图 1.3-8.1 迭代版

1. #include <stdio.h>

2. #include <time.h>

3. **clock_t** start, stop;

4. **void** check(**int**, **int**, **int** a[]);

5. **int** main()

6. {

7. **int** x, a[10];

8. scanf("%d", &x);

9. start = clock();

10. check(x, 0, a);

11. stop = clock();

12. printf("%f", (**double**)(stop - start)/CLK_TCK);

13. **return** 0;

14. }

(以上为 main.c)

1. #include <stdio.h>

2. **void** check(**int** x, **int** n, **int** a[])

3. {

```

4.  int t;
5.  for(int i = 0; i <= 9; i++)
6.  {
7.      t = 0;
8.      a[n] = i;
9.      if(n != 0)
10.     {
11.         for(int j = 0; j < n; j++)
12.             if(a[j] == a[n])
13.             {
14.                 t = 1;
15.                 break;
16.             }
17.         if(t)
18.             continue;
19.     }
20.     if(n < 9)
21.         check(x, n+1, a);
22.     if(n == 9)
23.     {
24.         if(x * (a[5]*10000+a[6]*1000+a[7]*100+a[8]*10+a[9]) == a[0]*100
           00+a[1]*1000+a[2]*100+a[3]*10+a[4])
25.             printf("%d%d%d%d%d/%d%d%d%d%d=%d\n", a[0], a[1], a[2]
           , a[3], a[4], a[5], a[6], a[7], a[8], a[9], x);
26.     }
27. }
28.}
29. //使用函数递归会使代码书写更为简洁,但是执行效率降低,所占用的储存
    空间更大

```

```
C:\WINDOWS\system32\cmd.exe
35
48265/01379=35
63945/01827=35
64295/01837=35
74865/02139=35
93485/02671=35
1. 295000请按任意键继续. . .
```

图 1.3-8.2 递归版

对比中可以很明显地发现递归的耗时明显长于迭代。这就说明在运算量较大时，为了保证程序能够计算完毕，我们可以考虑改进递归算法或者直接利用循环将之改写为迭代算法。

1.4 小结

本节实验加强了对顺序，分支，循环结构的掌握，初步探究了数组元素的存取操作，巩固了对状态的理解运用。并且为了更为高效的完成任务，自学了有关函数与递归的内容，并分析了递归与迭代的优劣。

程序改错环节，进一步加深了对优先级的概念的理解。进行了比较深入的非编译调试以寻找原本程序循环体中藏有的逻辑漏洞，并警醒了自己不能将逻辑判断的“==”写为赋值号“=”。

程序完善与修改替换的环节首先回顾了三大循环体的异同，并且对于 for 循环的出口条件进行了更为深入的分析。随后从寻找三位纯粹合数的程序设计之中感受到了利用潜在已知条件对结果进行约束以减小计算量的操作，并且运用了化归思想将新的问题转化成了之前处理过的问题。

程序设计的第 1 题强化了对 C 语言分支结构的 if-else 结构和 switch 结构的运用。第 2 题引入了数组进而对元素进行了按照下标次序的顺序存储，便于在后续程序段之中提取计算。第 3 题的两种思路，一种巩固了字符数组的存取操作，另一种则是回顾了所学习的利用 switch 结构进行状态判断而执行的不存储，接受后立刻判断立刻决定是否输出。第 6 题通过递归的方式进行了多次游戏的模拟，利用计算机的运算性能计算了所有可能的结果进而求得了准确的概率。第 7

题将求解孪生素数拆分为制作素数表以及判断是否为孪生素数两部分逐个分析处理，利用了化归的思想。最后的第 8 题，在完成了迭代的程序设计之后发现，代码量过大，并且重复较多，进而将其修改为递归的程序，并且加入了运行时间的分析，得出了在时间成本上，迭代由于递归的结论。

2 数组程序设计实验

2.1 程序改错与跟踪调试

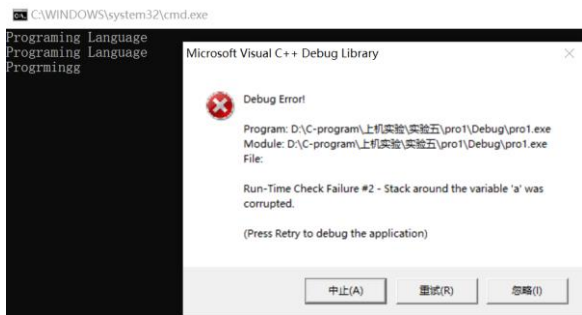


图 2.1-1

直接运行源程序时出现了越界的提示，检查发现第六行在初始化 b 数组时没有限定大小，就会直接被分配字符串长度加 1 作为数组大小。这是**第一处错误**。

修改后单步执行 `strcate()` 函数。

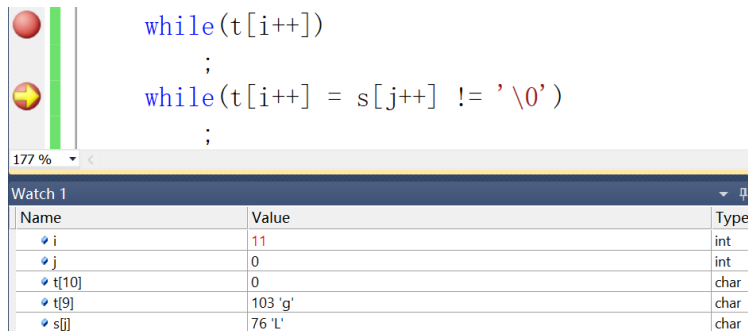


图 2.1-2

可以看出在执行完第一个循环时，i 值为 11，接下来就会把 `s[0]` 赋给 `t[11]`，而原本应赋给的 `t[10]` 却仍然是 `'\0'`。这是**第二处错误**，应当在 19，20 行之间加入使 i 减一的语句。**第三处错误**是我们之前详细探究过的优先级问题。

修改后单步执行 `strdelc()` 函数。

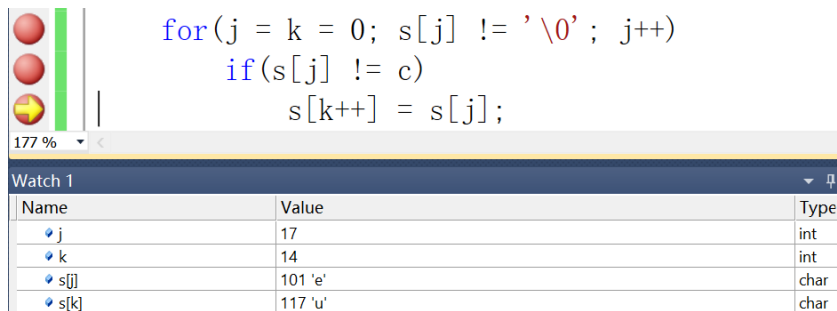


图 2.1-3

图 2.1-3 是在循环即将结束时的情况，做完当前行后会执行 `j++` 然后就会跳出循环，但是这样就会导致 `s[15]` 至 `s[17]` 没有被改变就会继续输出 `age` 然后遇到终止符结束。故应在第 29 行之后将 `s[15]` 设置为 `'\0'`。这是第四处错误。

源程序：

```
1. #include <stdio.h>
2. void strcat(char [], char []);
3. void strdelc(char [], char);
4. int main()
5. {
6.     char a[] = "Language", b[] = "Programing";
7.     printf("%s %s\n", b, a);
8.     strcat(b, a);
9.     printf("%s %s\n", b, a);
10.    strdelc(b, 'a');
11.    printf("%s\n", b);
12.    return 0;
13.}
14.
15. void strcat(char t[], char s[])
16. {
17.     int i = 0, j = 0;
18.     while(t[i++]
19.         ;
20.     while(t[i++] = s[j++] != '\0')
21.         ;
22. }
23.
24. void strdelc(char s[], char c)
25. {
26.     int j, k;
```

```

27.  for(j = k = 0; s[j] != '\0'; j++)
28.      if(s[j] != c)
29.          s[k++] = s[j];
30. }

```

(其中下划线处为第 1, 3 处错误)

修改后源程序:

```

1.  #include <stdio.h>
2.  void strcate(char [], char []);
3.  void strdelc(char [], char);
4.  int main()
5.  {
6.      char a[] = "Language", b[50] = "Programing";
7.      printf("%s %s\n", b, a);
8.      strcate(b, a);
9.      printf("%s %s\n", b, a);
10.     strdelc(b, 'a');
11.     printf("%s\n", b);
12.     return 0;
13. }
14.
15. void strcate(char t[], char s[])
16. {
17.     int i = 0, j = 0;
18.     while(t[i++])
19.         ;
20.     i--;
21.     while(t[i++] = s[j++])
22.         ;
23. }
24.

```



```
25. void strdelc(char s[], char c)
```

```
26. {
```

```
27.     int j, k;
```

```
28.     for(j = k = 0; s[j] != '\0'; j++)
```

```
29.         if(s[j] != c)
```

```
30.             s[k++] = s[j];
```

```
31.     s[k] = s[j];
```

```
32. }
```

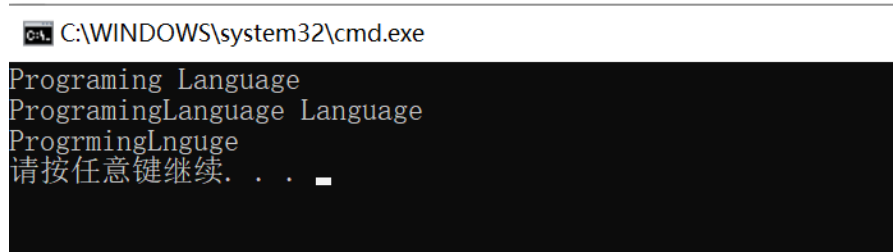


图 2.1-4 程序改错正确结果

2.2 程序完善与修改替换

(1)

本题要将后续有重复的字符消除掉，原方案是很自然的将所有曾经出现过多字符置为'\0'。但是这样操作在输出时，遇到第一个结束符就会停止输出，那么后面的没有重复的字符的输出同样会被截止，这就不符合我们的要求了。因此首先判断应当另外重组一遍数组下标，一种下标进度 r 用来扫描数组元素，若非结束符，就把下标进度 w 对应的元素置为 r 号元素，再使 w 自增。然后重新定义一种下标进度 $i = r + 1$ ，再用 i 扫描后续的元素，如有与 r 号元素相同的元素，则将其置为结束符。

完善后源程序（下划线处为填补内容）：

```
1. #include <stdio.h>
```

```
2. #include <string.h>
```

```
3. void RemoveDuplicate(char *s):
```

```

4.  int main()
5.  {
6.      char str[200];
7.      printf("Input strings, end of Ctrl+z\n");
8.      while(fgets(str, 200, stdin) != NULL)
9.      {
10.         RemoveDuplicate(str);
11.         printf("%s", str);
12.     }
13.     return 0;
14. }
15.
16. void RemoveDuplicate(char *s)
17. {
18.     int r, w, i, len;
19.     len = strlen(s);
20.     for(r = w = 0; r < len; r++)
21.     {
22.         if(s[r])
23.         {
24.             s[w++] = s[r];
25.             for(i = r + 1; i < len; i++)
26.                 if(s[i] == s[r])
27.                     s[i] = '\0';
28.         }
29.     }
30.     s[w] = '\0';
31. }

```

```
C:\WINDOWS\system32\cmd.exe
Input strings, end of Ctrl+z
aaaaaasssss dddadfdag
as dfg
asdfasdfasdfzxcvzxcvzxcv
asdfzxcv
Z
请按任意键继续. . .
```

图 2.2-1 完善（1）

然而在上述方法中程序的时间复杂度为 $O(n^2)$ ，现在要求提高运行的效率。分析程序可知， $s[i]$ 一旦被置为结束符之后，就完全可以不用再检验了，因为它不可能再是，重复的字符了。由此想到，只要我们能够知道有那些字符之前出现过了，就可以只通过一次扫描将整个工作做完。因此构造一个大小为 256 的标记数组，初始时元素全部置为 0。当扫描到某一个待处理的数组的元素 $s[r]$ 时，先以其 ASCII 码值为下标序号查看标记数组，若对应的元素为 0，则将其置为 1，再令 $s[w++] = s[r]$ ；反之若为 1 则不加处理继续扫描。由此就通过加大空间的开支（增加了内存的调用以储存标记数组）来减小了时间的花销，达到了提升效率的目的。

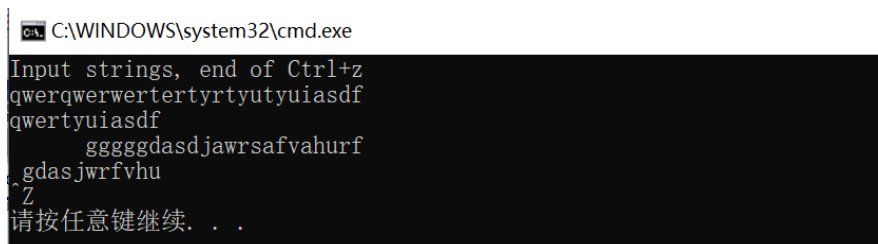
修改后源程序：

```
1. #include <stdio.h>
2. #include <string.h>
3. void RemoveDuplicate(char *s);
4. int tag[256] = {0};
5. int main()
6. {
7.     char str[200];
8.     printf("Input strings, end of Ctrl+z\n");
9.     while(fgets(str, 200, stdin) != NULL)
10.    {
11.        RemoveDuplicate(str);
12.        printf("%s", str);
13.    }
14.    return 0;
15.}
```

```

16.
17. void RemoveDuplicate(char *s)
18. {
19.     memset(tag, 0, sizeof(tag));
20.     int r, w, len;
21.     len = strlen(s);
22.     for(r = w = 0; r < len; r++)
23.     {
24.         int x = s[r];
25.         if(!tag[x])
26.         {
27.             tag[x] = 1;
28.             s[w++] = s[r];
29.         }
30.     }
31.     s[w] = '\0';
32. }

```



```

C:\WINDOWS\system32\cmd.exe
Input strings, end of Ctrl+z
qwerqwerwertertyrtyutyuiasdf
qwertyuasdf
gggggdasdjawrsafvahurf
gdasjwrfvhu
Z
请按任意键继续. . .

```

图 2.2-2 修改（1）

（2）

原题做法其实就是将现实生活中的做法换为了计算机语言表述。现实中完成该游戏时，淘汰的人就是要出圈，按次序站好（最后还要按次序报数）就可以了。所以我们可以压缩“圈内”数组，将淘汰的人按序加入到“出圈”数组之中就可以了。再根据已有的程序内容，只需我们将出圈和压缩的操作补上

即可。

由于我们是每报到 N 时就处理一次，所以肯定是当前报数的人应当被淘汰出圈。但是这个人对应的数组下标有两种可能，一种就是 $j-1$ ，另一种就是若出现了报到头的现象需要将 j 重新置为 0 ，而此时若恰好报到了 N ，就应当是“圈内”数组最后一个元素应当出去，对应的下标就是 $i-1$ 。因此第 18 行根据 j 是否为 0 的判断应当填入 $a[j-1]:a[i-1]$ 。

而压缩数组就是将出圈元素后的所有圈内元素向前移动一位，直观上就是 $a[k] = a[k+1]$ ，然而在调试时发现这样处理会在第一次进行压缩时出现数组越界的情况，因此我们可以通过取模的方法来避免这一危险操作。（最后一行的输出为运行时间）

完善后源程序（下划线处为填补内容）：

```
1. #include <stdio.h>
2. #define M 100
3. #define N 3
4. #include <time.h>
5. clock_t start, stop;
6. int main()
7. {
8.     int a[M], b[M];
9.     int i, j, k;
10.    for(i = 0; i < M; i++)
11.        a[i] = i + 1;
12.    start = clock();
13.    for(i = M, j = 0; i > 1; i--)
14.    {
15.        for(k = 1; k <= N; k++)
16.            if(++j > i - 1)
17.                j = 0;
18.        b[M-i] = j?a[j-1]:a[i-1];
19.        if(j)
```

```

20.     for(k = --j; k < i; k++)
21.         a[k] = a[(k+1) % M];
22.     }
23.     for(i = 0; i < M - 1; i++)
24.         printf("%6d", b[i]);
25.     printf("%6d\n", a[0]);
26.     stop = clock();
27.     printf("%f\n", (double)(stop - start) / CLK_TCK);
28.     return 0;
29. }

```

图 2.2-3 完善（2）

由于要不断地循环报数，所以这一部分的代码无法被修改以提高运行效率，但是压缩数组的步骤仍然可以通过标记数组的方法来进行修缮。这就相当于在循环过程中，这些被淘汰的人并没有出圈，而是每次轮到他们时不报数而是告诉大家他们已经出圈了。这样就省去了数组压缩的时间。

然而在实际操作中却发现原来的数据情形中，该方法并没有体现出明显的优越性，有时甚至会比前一种还要慢。我认为这可能是与每一次循环的时候由于他们没有出圈，所以仍会被扫描到，进而要进行判断有关。在加大了数据规模之后方才体现出其效率上的优势。当数据达到十万时后者比前者快接近 10 秒。

修改后的源程序：

```

1. #include <stdio.h>
2. #define M 100000

```

```

3. #define N 3
4. #include <time.h>
5. clock_t start, stop;
6. int main()
7. {
8.     int a[M];
9.     int i, j, k;
10.    for(i = 0; i < M; i++)
11.        a[i] = i + 1;
12.    start = clock();
13.    for(i = M, j = 0; i > 0; i--)
14.    {
15.        for(k = 1; k <= N; a[j++] && k++)
16.            if(j > M - 1)
17.                j = 0;
18.        if(j)
19.        {
20.            printf("%6d", a[j-1]);
21.            a[j-1] = 0;
22.        }
23.        else
24.        {
25.            printf("%6d", a[M-1]);
26.            a[M-1] = 0;
27.        }
28.    }
29.    stop = clock();
30.    printf("\n");
31.    printf("%f\n", (double)(stop - start) / CLK_TCK);
32.    return 0;

```

33.}

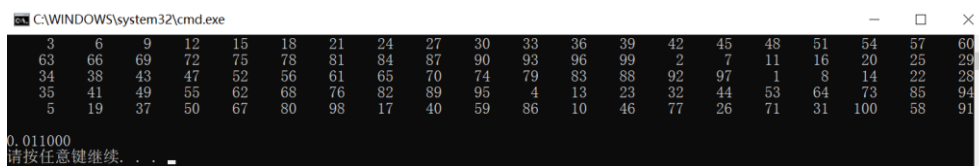


图 2.2-4 修改（2）

2.3 程序设计

(1)

依题意，即要将 32 位整数 x （假定 `int` 为 4 字节）转换为补码表示。我的第一想法较为复杂，没有考虑使用位运算的方法，而是按照取模的方式将 x 的绝对值的原码求出，再将其分类为正负，正数直接输出，而负数则按照转换为反码再转换为补码的方式。之后想到利用位运算可以大大简化程序。

思路 1 源码：

```
1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5.     int x;
6.     char a[32];
7.     while((scanf("%d", &x) != EOF))
8.     {
9.         memset(a, '0', sizeof(a));
10.        int x0 = (x >= 0)? x:-x;
11.        int i = 0;
12.        do{
13.            a[i++] = x0 % 2 + '0';
14.            x0 /= 2;
15.        }while(x0 > 0);
16.        //求|x|的原码
17.        if(x < 0)
```



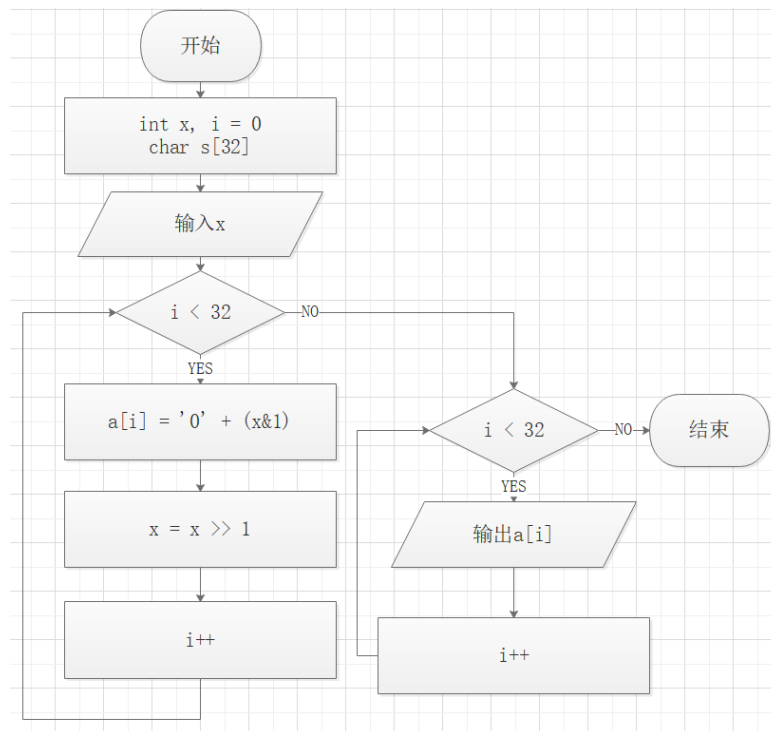
```

18.  {
19.      for(int i = 0; i < 32; i++)
20.          a[i] = !(a[i] - '0') + '0';
21.      //求 x 的反码
22.      if(a[0] == '0')
23.          a[0] = '1';
24.      else
25.      {
26.          int i = 0;
27.          do{
28.              a[i++] = (a[i] - '0' + 1) % 2 + '0';
29.          }while(a[i] == '1');
30.          if(i > 31)
31.              i = 31;
32.          a[i] = '1';
33.      }
34.  }
35.  //求 x 的补码
36.  for(int i = 31; i >= 0; i--)
37.      printf("%c", a[i]);
38.  printf("\n");
39.  }
40.  return 0;
41.}

```

思路 2：位运算

将数据按位与 1 实际上就是把数据在计算机内的表示的最低位取出。再将数据右移一位，继续按位与 1，直至取出最高位。



```

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5.     int x;
6.     char a[32];
7.     while((scanf("%d", &x) != EOF))
8.     {
9.         memset(a, '0', sizeof(a));
10.        for(int i = 0; i < 32; i++)
11.        {
12.            a[i] += (x & 1);
13.            x = x >> 1;
14.        }
15.        for(int i = 31; i >= 0; i--)
16.            printf("%c", a[i]);
17.        printf("\n");
  
```

```
18. }
19. return 0;
20. }
```

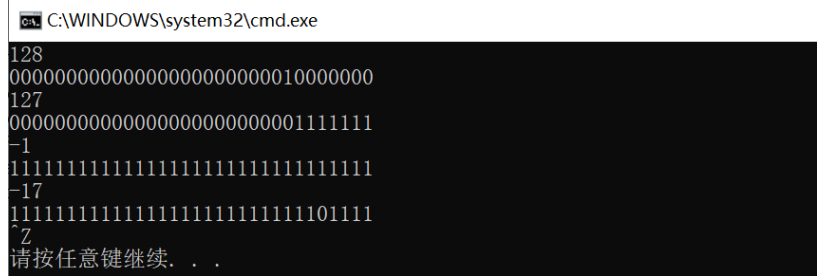
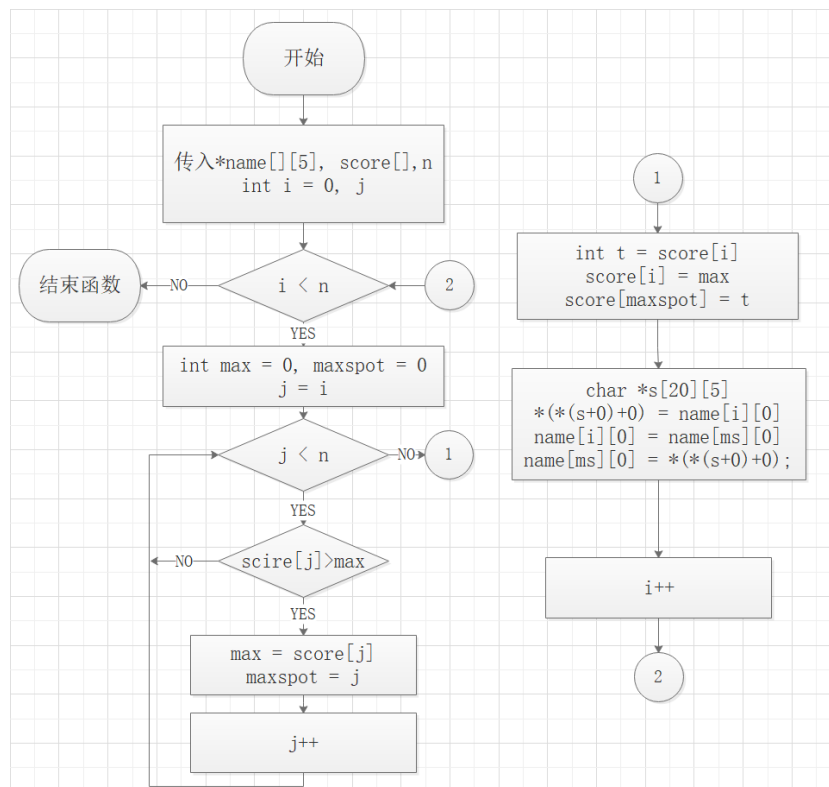


图 2.3-1 转换为二进制补码

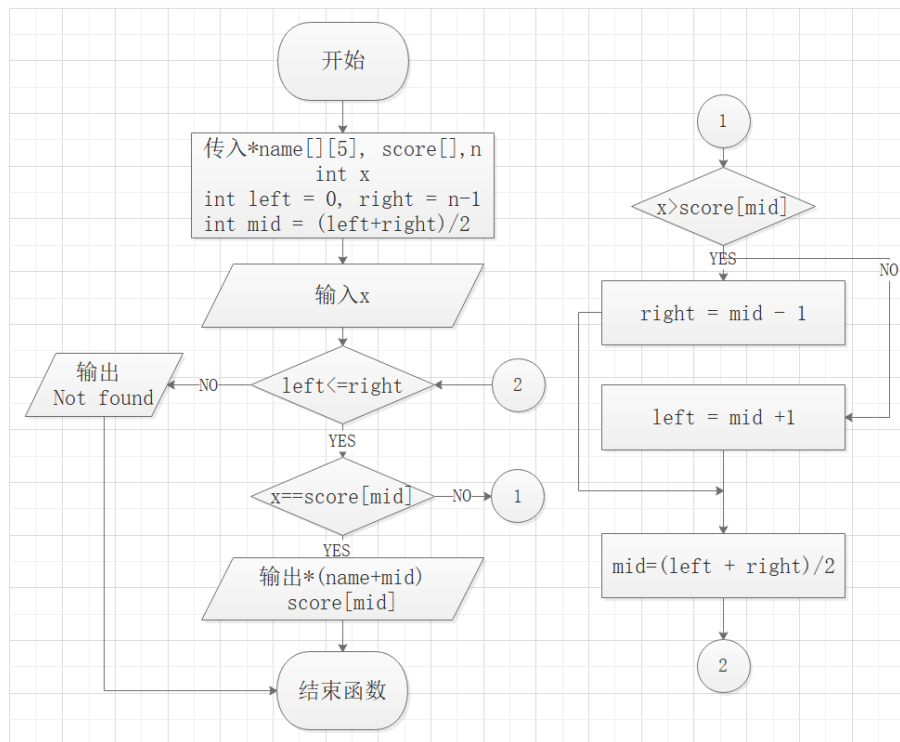
(2)

因为第三题是第二题的升级版因此这里直接合二为一。采用分治思想我们很简单的将任务分成四个函数：1.输入，2.排序，3.输出，4.查找。

在这里给出我们选择排序和二分查找的流程图:



选择排序



二分查找

1. `#include <stdio.h>`
2. `extern void input(char* name[][5], int score[], int n);`
3. `extern void sort(char* name[][5], int score[], int n);`
4. `extern void output(char* name[][5], int score[], int n);`
5. `extern void find(char* name[][5], int score[], int n);`

头文件

1. `#include "file.h"`
2. `int main()`
3. `{`
4. `int score[20], n;`
5. `char *name[20][5];`
6. `scanf("%d", &n);`
7. `input(name, score, n);`
8. `sort(name, score, n);`
9. `output(name, score, n);`
10. `find(name, score, n);`

```
11. return 0;
```

```
12. }
```

main 函数

```
1. #include "file.h"
```

```
2. void input(char* name[][5], int score[], int n)
```

```
3. {
```

```
4.     for(int i = 0; i < n; i++)
```

```
5.         scanf("%s %d", &name[i][0], &score[i]);
```

```
6. }
```

input 函数

```
1. #include "file.h"
```

```
2.
```

```
3. void sort(char* name[][5], int score[], int n)
```

```
4. {
```

```
5.     int max, ms, i;
```

```
6.     for(i = 0; i < n; i++)
```

```
7.     {
```

```
8.         max = 0, ms = 0;
```

```
9.         for(int j = i; j < n; j++)
```

```
10.        {
```

```
11.            if(score[j] > max)
```

```
12.            {
```

```
13.                max = score[j];
```

```
14.                ms = j;
```

```
15.            }
```

```
16.        }
```

```
17.     char *s[20][5];
```

```
18.     (*(s+0)+0) = name[i][0];
```

```
19.     name[i][0] = name[ms][0];
```

```
20.     name[ms][0] = (*(s+0)+0);
```

```

21.     int t = score[i];
22.     score[i] = max;
23.     score[ms] = t;
24. }
25. }

```

sort 函数

```

1. #include "file.h"
2.
3. void find(char* name[][5], int score[], int n)
4. {
5.     printf("find student who's score is:");
6.     int x, i, f = 0;
7.     scanf("%d", &x);
8.     int left = 0, right = n - 1, mid;
9.     while(left <= right)
10.    {
11.        mid = (left + right) / 2;
12.        if(score[mid] == x)
13.        {
14.            printf("%s %d\n", *(name+mid), score[mid]);
15.            f = 1;
16.            int k = 1;
17.            while(score[mid+k] == x)
18.            {
19.                printf("%s %d\n", *(name+mid+k), score[mid+k]);
20.                k++;
21.            }
22.            //若多人得分为 x，只用在 mid 两侧扫描即可确定最终有多少个
23.            k = 1;
24.            while(score[mid-k] == x)

```

```

25.     {
26.         printf("%s %d\n", *(name+mid-k), score[mid-k]);
27.         k++;
28.     }
29.     break;
30. }
31. else if(score[mid] > x)
32.     left = mid + 1;
33. else
34.     right = mid - 1;
35. }
36. if(!f)
37.     printf("Student of this score is not found\n");
38. }

```

find 函数

注：这里考虑了多个人得分为所要查找的分数的情况。根据数学逻辑我们明白相同分数的人的号码应当落在 mid 旁边，故只用从 mid 向两侧扫描即可。

```

1. #include "file.h"
2.
3. void output(char* name[][5], int score[], int n)
4. {
5.     printf("order after sorted:\n");
6.     for(int i = 0; i < n; i++)
7.         printf("%s %d\n", *(name+i), score[i]);
8.
9. }

```

output 函数

```

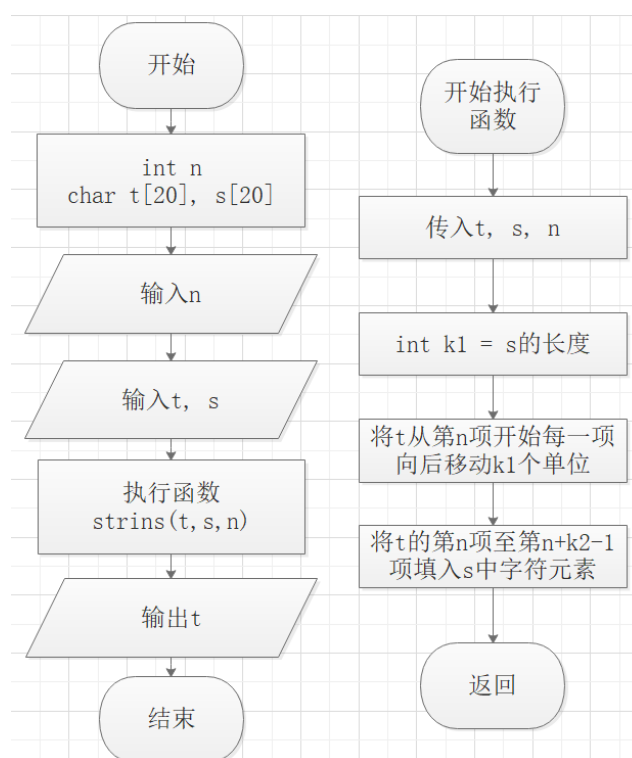
C:\WINDOWS\system32\cmd.exe
5
R.W 423
W.Y 412
T.T 234
Y.K 385
R.L 317
order after sorted:
R.W 423      W.Y 412      Y.K 385      R.L 317      T.T 234
find student who's score is:423
R.W 423
请按任意键继续. . .

```

图 2.3-2 学生成绩

(3)

字符插入的关键点是字符替换和整体移动。注意在整体向后移动时应当“从后向后移”否则会导致所有移动后的元素值均为第一个移动的元素值，从而导致元素的丢失。同理，整体向前移动应当“从前向前移”。



1. #include <stdio.h>
2. #include <string.h>
3. void strnins(char *, char *, int);
4. int main()
5. {


```

6.  int n;
7.  scanf("%d", &n);
8.  getchar();
9.  //利用 getchar()处理掉'\n'
10. char s[30], t[50];
11. fgets(t, 50, stdin);
12. fgets(s, 30, stdin);
13. //fgets()函数会计入'\n'
14. strnins(t, s, n);
15. printf("%s", t);
16. return 0;
17.}
18.
19. void strnins(char *t, char *s, int n)
20. {
21.     int i = n, j = 0;
22.     int k2 = strlen(s) - 1;
23.     //strlen(s)计入了'\n',需要减去
24.     int k1 = strlen(t) + k2;
25.     for(k1; k1 >= n + k2; k1--)
26.         t[k1] = t[k1 - k2];
27.     while(s[j] != '\n')
28.         t[i++] = s[j++];
29. }

```

C:\WINDOWS\system32\cmd.exe

```

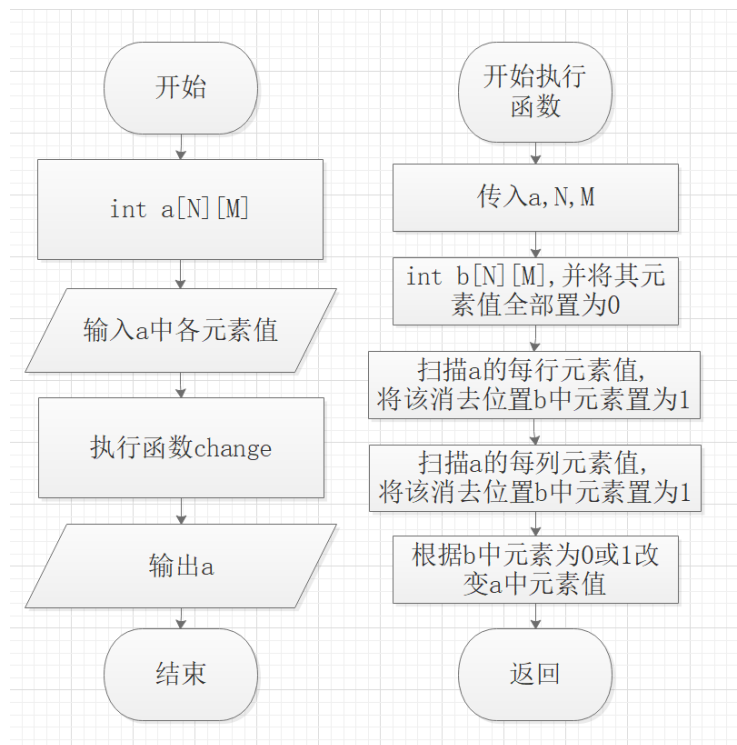
5
qwertyqwerty
asdf
qwertasdfqwerty
请按任意键继续. . .

```

图 2.3-3 字符插入

(4)

由于本题的主体是一个二维结构，我们很自然地想到使用一个二维数组去表示。但是不能够简单地直接对原数组进行消去，因为若是在扫描行的过程中就把数据清零了，可能会导致在扫描列的时候少消去一些数据。因此我们这里考虑添加一个等大小的参照数组，将改消去的数据位置上的元素置为 1，其余位置元素置为 0。然后再根据参照数组修改原数组就能够满足题目要求。



```
1. #include <stdio.h>
2. #define N 4
3. #define M 5
4. extern void enter(int (*a)[M]);
5. extern void change(int (*a)[M]);
6. extern void print(int (*a)[M]);
```

头文件

```
1. #include "file.h"
2. int main()
3. {
4.     int a[N][M];
```

```

5.  enter(a);
6.  change(a);
7.  print(a);
8.  return 0;
9.  }

```

main 函数

```

1.  #include "file.h"
2.  void enter(int (*a)[M])
3.  {
4.      for(int i = 0; i < N; i++)
5.          for(int j = 0; j < M; j++)
6.              scanf("%d", &a[i][j]);
7.  }

```

enter 函数

```

1.  #include "file.h"
2.  void change(int (*a)[M])
3.  {
4.      int b[N][M];
5.      int i, j;
6.      for(i = 0; i < N; i++)
7.          for(j = 1; j < M; j++)
8.          {
9.              int count = 1;
10.             while(a[i][j] == a[i][j-1] && j < M)
11.             {
12.                 j++;
13.                 count++;
14.             }
15.             if(count >= 3)
16.             {

```

```

17.     for(int k = j - count; k < j; k++)
18.         b[i][k] = 1;
19.     }
20. }
21. for(i = 0; i < M; i++)
22.     for(j = 1; j < N; j++)
23.     {
24.         int count = 1;
25.         while(a[j][i] == a[j-1][i] && j < N)
26.         {
27.             j++;
28.             count++;
29.         }
30.         if(count >= 3)
31.         {
32.             for(int k = j - count; k < j; k++)
33.                 b[k][i] = 1;
34.         }
35.     }
36. for(i = 0; i < N; i++)
37.     for(j = 0; j < M; j++)
38.         if(b[i][j] == 1)
39.             a[i][j] = 0;
40. }

```

change 函数

```

1. #include "file.h"
2. void print(int (*a)[M])
3. {
4.     for(int i = 0; i < N; i++)
5.     {

```

```

6.     for(int j = 0; j < M; j++)
7.         printf("%d\t", a[i][j]);
8.     printf("\n");
9. }
10.}

```

print 函数

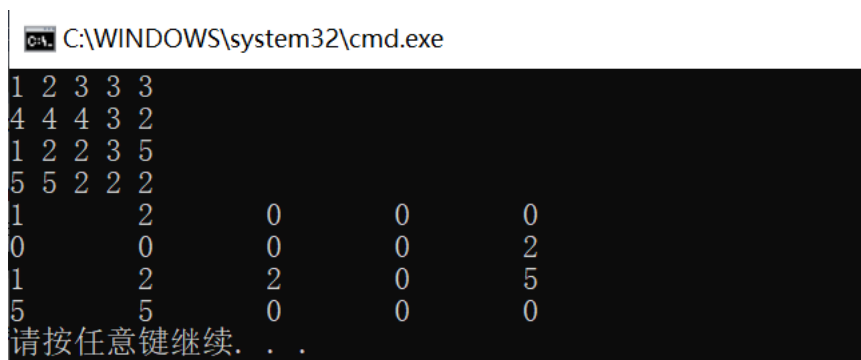


图 2.3-4 数字消除

（上一组数据为输入，下一组数据为输出）

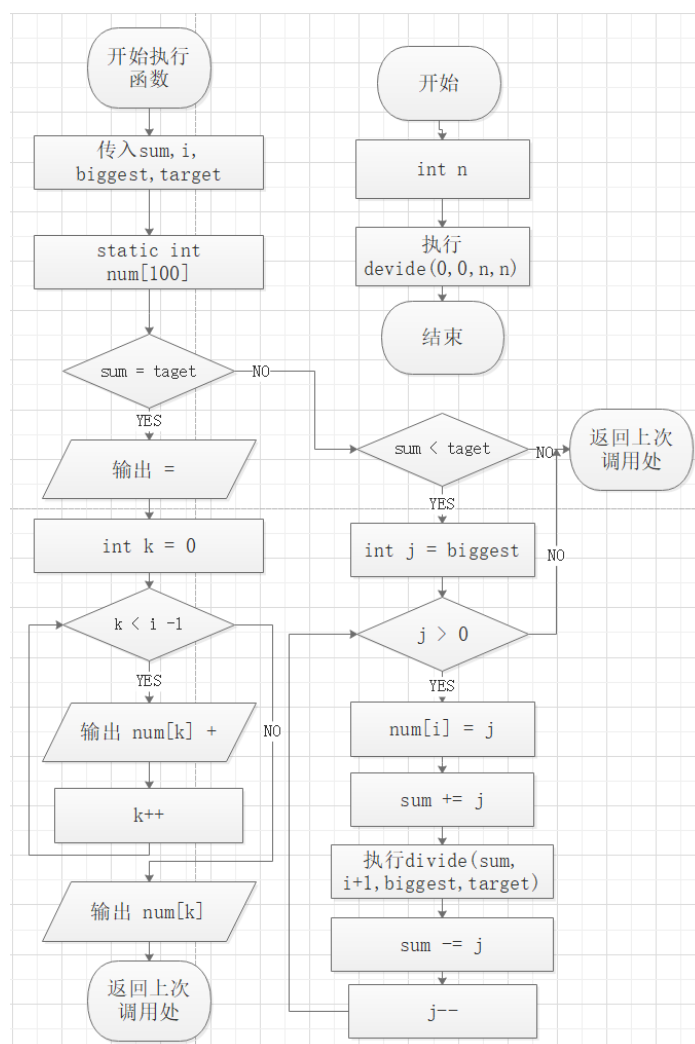
在这里我们讨论一下程序的修改问题。可以发现进行行扫描和列扫描的代码基本上没有差异，唯一差别就在于交换了行列序号，进而联想到可能可以将这一段改写为函数。但是基于 C 语言数组元素的储存方式，可以较为简便地整理出进行行扫描的函数，但是如果要将其应用到列扫描时，却要将列的元素取出重置为行数组，这一操作的代码量与直接再扫描列并无本质差别，反而影响了程序的整体可读性，再调用函数又将降低程序运行的效率（虽然在本例中不会明显显现），因此最终没有考虑像之前 1.3-（8）中一样再给出增添函数之后的版本。

（5）

最开始着眼本题时感觉是一道比较适合使用递归的方法去解决的问题，但仔细考虑，似乎并没有想象中的简单易行。首先是拆分原数 **target**，得到 **a0** 与 **a1**，之后接着要拆分 **a1** 得到 **b0** 与 **b1**，如此进行递归。同时要注意到每次拆分的 **x0** 均应当大于等于 **x1**。感觉拆分起来有一定的复杂性，一时间没有想到较为简便的符合递归的算法。因此反向考虑，将拆解问题转化为求和问题，只要

满足每次递归的“最大拆解数”的和等于 **target** 即可。

基于以上分析，考虑使用的递归函数要传入的参数有最终目标数 **target**，当前拆解最大数值 **biggest**，递归次数 **i**（方便将每次拆解出来的最大数存储以及最后的输出），以及拆解最大数之和 **sum**。



```

1. #include <stdio.h>
2. void divide(int sum, int i, int biggest, int target);
3. int main()
4. {
5.     int n;
6.     scanf("%d", &n);
7.     divide(0, 0, n, n);
8.     return 0;

```

```

9. }
10.
11. void divide(int sum, int i, int biggest, int target)
12. {
13.     static int num[100];
14.     if(sum > target)
15.         return;
16.     else if(sum == target)
17.     {
18.         printf(" =");
19.         int k = 0;
20.         for(; k < i - 1; k++)
21.             printf(" %d +", num[k]);
22.         printf(" %d\n", num[k]);
23.     }
24.     else
25.     {
26.         int j = biggest;
27.         //保证后面的加数不大于前面的被加数
28.         for(; j > 0; j--)
29.         {
30.             num[i] = j;
31.             sum += j;
32.             divide(sum, i + 1, j, target);
33.             sum -= j;
34.             //使当前和恢复加 j 前状态
35.         }
36.     }
37. }

```

可以比较清晰地看出这里的算法思路，就是从 **biggest** 开始给 **sum** 加上去，然后当 **sum** 仍然小于 **target** 的时候继续执行 **divide** 函数，直至 **sum** 大于或等于 **target**，分别对应回到上次调用处和将结果输出再回到上次调用处。然后因为我们仍然处在 **j** 的循环内还要继续给原来的 **sum** 加上下一个 **j**，所以我们需要将 **sum** 还原回去，再继续下一轮的循环。

接下来是在课上单步执行调试时发现的可做小改进的位置。

我们发现在取 **j** 时会进行一些不必要的计算，比如当 **target=6**，**biggest=5** 的时候我们完全没有必要进行 **j=5, 4, 3, 2** 的计算调用，因为如此操作 **sum** 一定会大于 **target**，因此我们可以将程序修改如下：

当 **biggest!=target** 时（否则 **j** 取 **biggest**）判断 **biggest** 与 **target-sum** 的大小并取其中较小的那个以减小计算量。

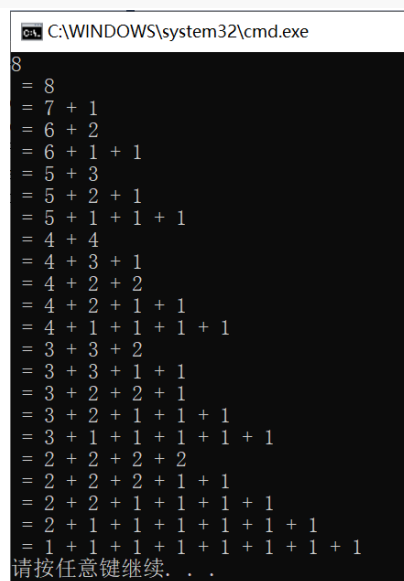
```
1. #include <stdio.h>
2. void divide(int sum, int i, int biggest, int target);
3. int main()
4. {
5.     int n;
6.     scanf("%d", &n);
7.     divide(0, 0, n, n);
8.     return 0;
9. }
10.
11. void divide(int sum, int i, int biggest, int target)
12. {
13.     static int num[100];
14.
15.     if(sum == target)
16.     {
17.         printf(" =");
18.         int k = 0;
```



```

19.     for(; k < i - 1; k++)
20.         printf(" %d +", num[k]);
21.     printf(" %d\n", num[k]);
22. }
23. else
24. {
25.     int j = (biggest==target || biggest < (target-sum))? biggest:(target-
        sum);
26.     //保证后面的加数不大于前面的被加数且加上后不会超过 target
27.     for(; j > 0; j--)
28.     {
29.         num[i] = j;
30.         sum += j;
31.         divide(sum, i + 1, j, target);
32.         sum -= j;
33.         //使当前和恢复加 j 前状态
34.     }
35. }
36.}

```



```

C:\WINDOWS\system32\cmd.exe
8
= 8
= 7 + 1
= 6 + 2
= 6 + 1 + 1
= 5 + 3
= 5 + 2 + 1
= 5 + 1 + 1 + 1
= 4 + 4
= 4 + 3 + 1
= 4 + 2 + 2
= 4 + 2 + 1 + 1
= 4 + 1 + 1 + 1 + 1
= 3 + 3 + 2
= 3 + 3 + 1 + 1
= 3 + 2 + 2 + 1
= 3 + 2 + 1 + 1 + 1
= 3 + 1 + 1 + 1 + 1 + 1
= 2 + 2 + 2 + 2
= 2 + 2 + 2 + 1 + 1
= 2 + 2 + 1 + 1 + 1 + 1
= 2 + 1 + 1 + 1 + 1 + 1 + 1
= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
请按任意键继续...

```

图 2.3-5 整数的划分

(6)

与上一题类似，拿到迷宫问题第一感觉就是递归地走下一步，如果发现不符合要求就回退到上一步。然后在实现过程中发觉应当先判断再走，比先走在判断再可能回退更为优化，因为这样就可以略去回退的步骤，简少了递归函数的调用次数并降低了编写的难度，就在编程过程中针对这一问题转换了编写思路。

迷宫问题中的步法较为简单，只有上下左右四种可能，因此可以利用两个一维数组 **dx** 与 **dy** 的排列组合将之表达出来。每一次递归函数被执行时这四种可能下一步的都要被判断是否合理。

编写程序的核心在于判断走出的下一步是否合理，而进行这一判断的关键有 3 点：

1) 下一步走出后是否出界。这一点检查起来也较为方便，只用传入迷宫的大小就可以判断是否出界了。

2) 下一步走到的位置是否为墙壁。检查起来同样比较方便，传入了储存迷宫的二维数组，判断下一步对应的位置上的元素是否为 1（这里将元素为 1 记作墙壁）就可以了。

3) 下一步走的是否是曾经到达过的位置，这一点很关键（若被忽略则可能会导致递归函数陷入死循环式的重复调用）而且相较前两点更容易被忽略。判断这一问题我们需要将之前抵达过的每一步都储存起来，然后利用循环判断下一步是否与之前的任意一步相同。

在走完下一步后就执行下一次递归，判断是否抵达终点，若是则输出从起点开始经过的每一步，否则继续执行递归判断下一步是否合理。

```
1. #include <stdio.h>
2. #define N 6
3. #define M 10
4. int dx[] = {0, 1, -1, 0};
5. int dy[] = {1, 0, 0, -1};
6. int pos[100][2];
7. void step(int i, int (*a)[M]);
8. void print_step(int n, int (*p)[2]);
```

```

9.  int count;
10. int main()
11. {
12.     int a[N][M];
13.     for(int i = 0; i < N; i++)
14.         for(int j = 0; j < M; j++)
15.             scanf("%d", &a[i][j]);
16.     pos[0][0] = pos[0][1] = 0;
17.     step(0, a);
18.     if(!count)
19.         printf("No available route found");
20.     return 0;
21. }
22.
23. void step(int i, int(*a)[M])
24. {
25.     if(pos[i][0] == N-1 && pos[i][1] == M-1)
26.     {
27.         count++;
28.         print_step(i, pos);
29.         return;
30.     }
31.     int j;
32.     for(j = 0; j < 4; j++)
33.     {
34.         int k, t = 1;
35.         if(!(pos[i][0] + dx[j] >= 0 && pos[i][1] + dy[j] >= 0 && pos[i][0] + dx[j]
            <= N-1 && pos[i][1] + dy[j] <= M-1)) //判断是否出界
36.             continue;
37.         else

```

```

38.     {
39.         for(k = 0; k <= i && t; k++)
40.         {
41.             if(pos[i][0] + dx[j] == pos[k][0] && pos[i][1] + dy[j] == pos[k][1])
                //判断是否到达过该位置
42.                 t = 0;
43.             else if(a[pos[i][0] + dx[j]][pos[i][1] + dy[j]]) //判断是否为墙壁
44.                 t = 0;
45.         }
46.         if(t)
47.         {
48.             pos[i+1][0] = pos[i][0] + dx[j];
49.             pos[i+1][1] = pos[i][1] + dy[j];
50.             step(i+1, a);
51.         }
52.         else
53.             continue;
54.     }
55. }
56.}
57.
58. void print_step(int n, int (*p)[2])
59. {
60.     printf("Num %d:\n", count);
61.     for(int i = 0; i < n; i++)
62.         printf("(%d,%d) -> ", p[i][0], p[i][1]);
63.     printf("(%d,%d)\n", N-1, M-1);
64. }

```

```
C:\WINDOWS\system32\cmd.exe
0 1 1 1 1 1 1 1 1 1
0 1 0 0 1 0 0 0 0 1
0 0 0 1 1 0 0 1 0 1
0 0 0 1 1 0 0 1 0 1
0 1 1 0 0 0 1 1 0 1
0 0 0 0 1 0 0 0 0 1
1 1 1 1 1 1 1 1 0 0
Num 1:
(0,0) -> (1,0) -> (2,0) -> (3,0) -> (4,0) -> (4,1) -> (4,2) -> (4,3) -> (3,3) -> (3,4) -> (3,5) -> (4,5) -> (4,6)
-> (4,7) -> (4,8) -> (5,8) -> (5,9)
Num 2:
(0,0) -> (1,0) -> (2,0) -> (3,0) -> (4,0) -> (4,1) -> (4,2) -> (4,3) -> (3,3) -> (3,4) -> (3,5) -> (2,5) -> (2,6)
-> (1,6) -> (1,7) -> (1,8) -> (2,8) -> (3,8) -> (4,8) -> (5,8) -> (5,9)
Num 3:
(0,0) -> (1,0) -> (2,0) -> (3,0) -> (4,0) -> (4,1) -> (4,2) -> (4,3) -> (3,3) -> (3,4) -> (3,5) -> (2,5) -> (1,5)
-> (1,6) -> (1,7) -> (1,8) -> (2,8) -> (3,8) -> (4,8) -> (5,8) -> (5,9)
请按任意键继续. . .
```

图 2.3-6 迷宫问题

2.4 小结

在本节的实验中，主要学习了数组的储存与应用。

在修改程序的例题中，明确了初始化数组时指定数组大小与不指定的差异，并且感受到了字符数组中有效内容之后如果不加入结束符可能会导致的乱码问题以及如果在操作中提前将某位置的元素置为结束符所导致的字符串输出提前终止的问题。

在改写程序中初步接触了算法的优化，感受了时间复杂度的简单计算并了解掌握了通过增大空间的开支引入一个标记数组以提高程序效率的方法以及其背后的利用空间换时间的思想。并且在观察到实际处理与处理前的设想的出入，然后给出了自己的猜想与分析。尽管目前的程序优化其实仍然停留在表面层次，但是也为日后的算法与数据结构的学习打下了一定的基础。

在接下来的程序设计问题中，首先学习了数字字符串与数值的转换并在优化时巩固了位运算的思维与使用方法。然后初步探索了如何将数组作为参数传入函数，为此查阅了有关数组指针的资料。自己想到了使用类似于选择排序法的方法（在余下元素找到最大或最小的哪一个，与指定位置处的元素进行交换）对数组元素进行重排。在第 5 道实验题中思考探究了自某项开始整体移动数组元素的方法，并进行了归纳以便于记忆。第 6 道实验继续巩固了对标记法的理解，使用了二维标记数组作为参照数组去控制原数组的。第 9 和 11 题则加深了对于递归程序的理解，并分析比较了先判断再执行与先执行再判断的优劣。其中第 9 题的设计中体现了逆向思维的使用，将划分转化为求和。

3 结构与联合实验

s3.1 表达式求值的程序验证

分析如下：

1. `(++p)->x`

开始时 `p` 指向的是 `a` 的起始地址，执行前缀自增操作之后 `p` 指向 `a` 的第一个元素（这里数组元素从 0 开始计入）`a[1]`，接下来通过成员选择操作，选择 `a[1]` 中的 `x` 成员，也就是 100。

2. `p++,p->c`

开始时 `p` 指向的是 `a` 的起始地址，然后执行后缀自增操作，遇到“,”时生效，使得 `p` 指向 `a[1]`，再通过成员选择操作选择 `a[1]` 中的 `c` 也就是“B”。最后由于含逗号运算符的表达式的是最右侧的表达式，故所求为“B”。

3. `*p++->t,*p->t`

由于*的优先级低于自增，所以左侧表达式应先执行 `p++`，再取此时 `p` 指向的结构变量元素的成员 `t` 所指向的字符变量元素。但是由于是后缀表达式，只有在遇到序列点时才会生效，故最终左侧的效果是取得“U”，并使得 `p` 指向了 `a[1]`。之后取 `a[1]` 的成员 `t`，再取 `t` 所指向的字符，最终表达式的结果应为“x”。

4. `*(++p)->t`

由于*的优先级最低，该表达式先执行 `(++p)` 使得 `p` 指向 `a[1]`，再取 `t`，最后再取 `t` 所指向的字符变量也即是“x”。

5. `*++p->t`

由于 `->` 优先级最高，该表达式先执行取 `p` 指向的结构变量 `a[0]` 中的成员 `t`，再对 `t` 执行前缀自增操作，时期指向元素 `u[1]`，最后*操作取得 `u[1]` 也即是“U”。

6. `++*p->t`

同 5 中分析，应当先执行取 `p` 指向的结构变量 `a[0]` 中的成员 `t`，再取得此时 `t` 指向的元素 `u[0]`——“U”，然后再使其自增，得到最终结果“V”。

验证程序：

```
1. #include <stdio.h>
2. char u[] = "UVWXYZ", v[] = "xyz";
```

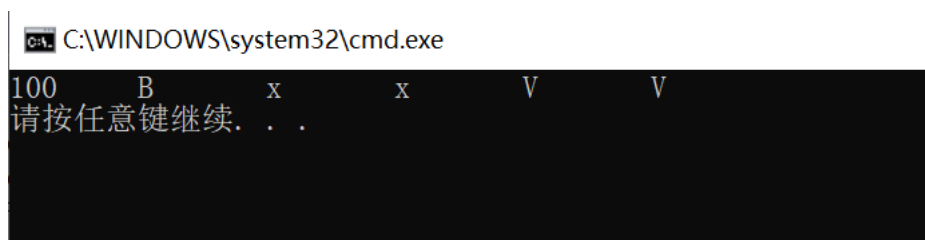
```

3. struct T{
4.     int x;
5.     char c;
6.     char *t;
7. }a[] = {{11, 'A', u}, {100, 'B', v}}, *p = a;
8. int main()
9. {
10.    printf("%d\t", (++p)->x);
11.    --p;
12.    printf("%c\t", (p++, p->c));
13.    --p;
14.    printf("%c\t", (*p++->t, *p->t));
15.    --p;
16.    printf("%c\t", *(++p)->t);
17.    --p;
18.    printf("%c\t", *++p->t);
19.    --p->t;
20.    printf("%c\n", ++*p->t);
21.}

```

其中 11, 13 等行的操作是为了还原指向关系。

验证结果：



```

C:\WINDOWS\system32\cmd.exe
100 B x x V V
请按任意键继续. . .

```

图 3.1-1 表达式验证

表 3.1-1 表达式求值及验证

序号	表达式	计算值	验证值	正确与否
1	<code>(++p)->x</code>	100	100	√
2	<code>p++,p->c</code>	B	B	√
3	<code>*p++->t,*p->t</code>	x	x	√
4	<code>*(&p)->t</code>	x	x	√
5	<code>*++p->t</code>	V	V	√
6	<code>++*p->t</code>	V	V	√

3.2 源程序修改替换

(1) 修改

在我的编译器环境下直接执行结果是没有任何有效输出,通过分析问题应当是出在了给函数传入的参数的类型上。如果要在函数中改变一个函数外的变量的值,唯一的方法就是通过传址,以指向该变量的指针变量类型为形参。具体到这里,我们要改变的是头指针,那么就应当传入的就是头指针的地址(或指向头指针的函数)。所以改法很简单,只用将传参的过程修改一下就可以了。

然后我们会发现原题的编程存在没有释放动态申请的内存空间的问题。根据我在 CSDN 平台上查找到的资料,在我们的程序需要动态申请空间时操作系统只是会给我们分配一个虚拟的地址空间:把应用程序的虚拟地址空间映射到真实的物理地址(或者磁盘上的分页文件)。也就是说,在这一理论下,不管用户程序怎么 `malloc`,在进程结束的时候,其虚拟地址空间就会被直接销毁,操作系统只需要在进程结束的时候,让内存管理模块把分页文件中与此进程相关的记录全部删除,标记为“可用空间”,就可以使所有申请的内存都一次性地回收。但是基于一个良好的编程习惯,为我们自己要跟操作系统直接“博弈”打下基础,仍应添加上 `free()` 的语句。

修改后的程序:

```
1. #include <stdio.h>
2. #include <stdlib.h>
```



```

3. struct s_list{
4.     int data;
5.     struct s_list *next;
6. };
7. void create_list(struct s_list **headp, int *p);
8. int main()
9. {
10.    struct s_list *head = NULL, *p;
11.    int s[] = {1, 2, 3, 4, 5, 6, 7, 8, 0};
12.    create_list(&head, s);
13.    p = head;
14.    while(p)
15.    {
16.        printf("%d\t", p->data);
17.        p = p->next;
18.    }
19.    printf("\n");
20.    while(head)
21.    {
22.        p = head->next;
23.        free(head);
24.        head = p;
25.    }
26.    free(head);
27.    //以上这一部分是自己添加的释放内存部分
28.    return 0;
29.}
30.
31. void create_list(struct s_list **headp, int *p)
32. {

```

```

33.  struct s_list *loc_head = NULL, *tail;
34.  if(!p[0])
35.      ;
36.  else
37.  {
38.      loc_head = (struct s_list*)malloc(sizeof(struct s_list));
39.      loc_head->data = *p++;
40.      tail = loc_head;
41.      while(*p)
42.      {
43.          tail->next = (struct s_list*)malloc(sizeof(struct s_list));
44.          tail = tail->next;
45.          tail->data = *p++;
46.      }
47.      tail->next = NULL;
48.      *headp = loc_head;
49.      int i = 0;
50.      i++;
51.  }
52.}

```

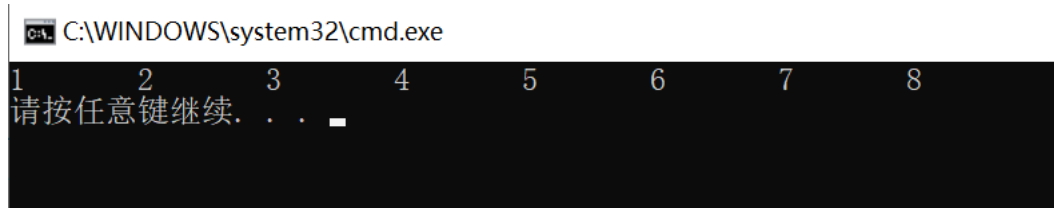


图 3.2-1 先进先出链表

(2) 替换程序构建后进先出链表

因为在建立先进先出的链表时，我们需要通过对尾结点的后继结点指针进行更改所以才需要引入 **tail** 这个工作指针，而由于在建立后进先出的链表时最开始建立的先建结点一定是尾结点，所有的后建结点都是自然地指向它的先建

结点的，所以不用再引入 **tail** 指针来指向尾结点了。

下面分析后进先出链表设计的关键：

因为头结点永远是后建结点，所以真正的建立过程实际上是不断地去执行头插入的操作，因此我们只用建立一个 ***cur** 结点，为它动态申请一个对应的内存空间，然后使它的后继结点指针指向头结点，给它的数据域赋值，再将头结点更新为它就可以了。

具体实现：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. typedef struct list{
5.     int data;
6.     struct list *next;
7. }list;
8.
9. void create(list **, int *);
10.
11. int main()
12. {
13.     int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 0};
14.     list *head = NULL, *p;
15.     create(&head, a);
16.     p = head;
17.     while(p)
18.     {
19.         printf("%d\t", p->data);
20.         p = p->next;
21.     }
22.     printf("\n");
23.     while(head)
```

```

24. {
25.     p = head->next;
26.     free(head);
27.     head = p;
28. }
29. free(head);
30. return 0;
31. }
32.
33. void create(list **headp, int *p)
34. {
35.     list *loc_head = NULL;
36.     if(!*p)
37.         ;
38.     else
39.     {
40.         loc_head = (list *)malloc(sizeof(list));
41.         loc_head->data = *p++;
42.         loc_head->next = NULL;
43.         while(*p)
44.         {
45.             list *cur = (list *)malloc(sizeof(list));
46.             //建立临时结点
47.             cur->data = *p++;
48.             //给临时结点数据域赋值
49.             cur->next = loc_head;
50.             //使临时结点的后继结点为当前的头结点
51.             loc_head = cur;
52.             //更新头结点
53.         }

```

```
54. }  
55. *headp = loc_head;  
56. }
```

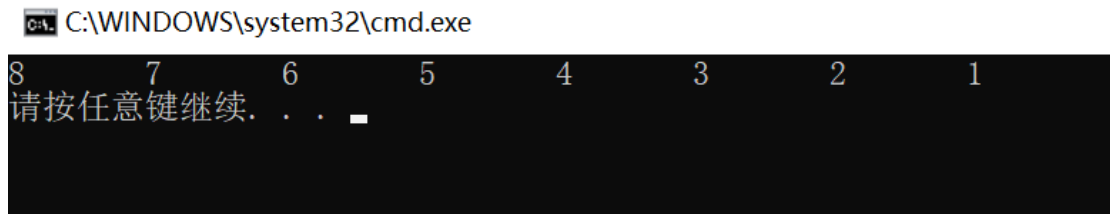


图 3.2-2 后进先出链表

3.3 程序设计

(1)

我对这一题的目的的理解是要将某一个无符号字符变量的低字节的所有位中的值为 1 的位的序列输出。那么比较自然的想法就是通过设立一个 union 结构，使字段结构变量 struct bits b 和一个无符号 char 型数据 ch 共用一个储存单元，然后根据字段结构成员 bit_i(i = 0,1,2...,7)的取值是否为 1，去调用执行直线建立好的函数表（函数指针数组）中的对应的函数就可以达到想要的目的了。

```
1. #include <stdio.h>  
2. struct bits{  
3.     unsigned char bit0:1;  
4.     unsigned char bit1:1;  
5.     unsigned char bit2:1;  
6.     unsigned char bit3:1;  
7.     unsigned char bit4:1;  
8.     unsigned char bit5:1;  
9.     unsigned char bit6:1;
```

```

10. unsigned char bit7:1;
11. };
12. union ch_bits{
13.     unsigned char ch;
14.     struct bits b;
15. };
16. void f0(int b)
17. {
18.     printf("the function0 %d is called!\n", b);
19. }
20. void f1(int b)
21. {
22.     printf("the function1 %d is called!\n", b);
23. }
24. void f2(int b)
25. {
26.     printf("the function2 %d is called!\n", b);
27. }
28. void f3(int b)
29. {
30.     printf("the function3 %d is called!\n", b);
31. }
32. void f4(int b)
33. {
34.     printf("the function4 %d is called!\n", b);
35. }
36. void f5(int b)
37. {
38.     printf("the function5 %d is called!\n", b);
39. }

```

```

40. void f6(int b)
41. {
42.     printf("the function6 %d is called!\n", b);
43. }
44. void f7(int b)
45. {
46.     printf("the function7 %d is called!\n", b);
47. }
48. int main()
49. {
50.     union ch_bits cb;
51.     void (*p_fun[8])(int b);
52.     p_fun[0] = f0;
53.     p_fun[1] = f1;
54.     p_fun[2] = f2;
55.     p_fun[3] = f3;
56.     p_fun[4] = f4;
57.     p_fun[5] = f5;
58.     p_fun[6] = f6;
59.     p_fun[7] = f7;
60.     while((scanf("%c", &cb.ch) != EOF))
61.     {
62.         if(cb.b.bit0)
63.             p_fun[0](cb.b.bit0);
64.         if(cb.b.bit1)
65.             p_fun[1](cb.b.bit1);
66.         if(cb.b.bit2)
67.             p_fun[2](cb.b.bit2);
68.         if(cb.b.bit3)
69.             p_fun[3](cb.b.bit3);

```

```

70.     if(cb.b.bit4)
71.         p_fun[4](cb.b.bit4);
72.     if(cb.b.bit5)
73.         p_fun[5](cb.b.bit5);
74.     if(cb.b.bit6)
75.         p_fun[6](cb.b.bit6);
76.     if(cb.b.bit7)
77.         p_fun[7](cb.b.bit7);
78.     getchar();
79.         //用 getcahr()处理'\n'
80. }
81. return 0;
82.}

```

```

C:\WINDOWS\system32\cmd.exe
A
the function0 1 is called!
the function6 1 is called!
B
the function1 1 is called!
the function6 1 is called!
C
the function0 1 is called!
the function1 1 is called!
the function6 1 is called!
D
the function2 1 is called!
the function6 1 is called!
Z
请按任意键继续. . .

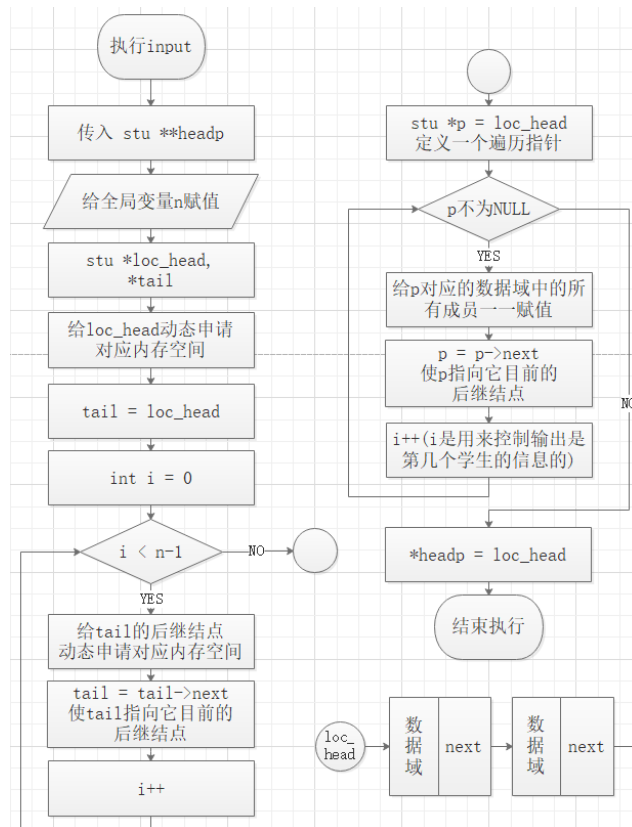
```

图 3.3-1 数段结构应用

可以很清晰的看出从 A 到 B 直到 D，每一个数据之间的差距都是 1，这就与实际数据的变化相吻合了。

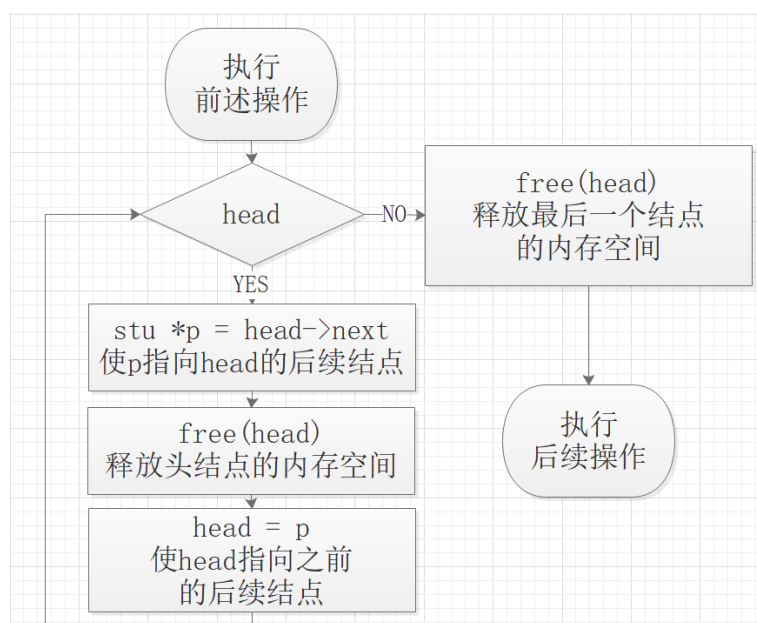
(2)

本题要求通过建立单向链表来储存学生的基本信息及各项成绩。这里单论输入，输出，修改，求平均值都是比较常规的操作，在往期的实验中都有涉及。而唯一的区别就在于链表的设计，这里我们就用流程图的形式，展示将建立链表与数据录入结合的函数的设计思路。



上页的流程图左侧为构建结点数为 n 的链表，右侧则是对它的数据域进行赋值。仍然要注意的使传入的参数，在主函数中应传入一个 stu^* 类型的变量的地址。右下角给出的是链表的结构示意图，要注意的是链尾 tail 结点的后续结点指针 $\text{next} = \text{NULL}$ 。

本章的程序之中我都加入了释放的代码，在这里就展示一下释放操作的流程图。



```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. typedef struct stu{
5.     int english;
6.     int math;
7.     int physics;
8.     int c;
9.     char num[5];
10.    char name[10];
11.    struct stu *next;
12. }stu;
13. typedef struct aver{
14.     double s_aver;
15.     struct aver *next;
16. }aver;
17.
18. int n;
19. void input(stu**);
20. void output(stu*);
21. void change(stu**, int);
22. void c_aver(stu*, int, aver**);
23. void print_all(stu*, aver*);
24.
25. int main()
26. {
27.     stu *head;
28.     aver *head1;
29.     int i, t = 0;
```

```

30. printf("1\tinput the information of each student\n");
31. printf("2\toutput the information of each student\n");
32. printf("3\tchange a student's information\n");
33. printf("4\tcollect the average score of each student\n");
34. printf("5\toutput all of the information of the student\n");
35. printf("0\tend of process\n");
36. while((scanf("%d", &i)) != 0)
37. {
38.     if(i)
39.         t = 1;
40.     //防止出现没有执行任何指令直接退出导致的 free()出
    现 Debug error
41.     switch(i)
42.     {
43.         case 1:input(&head);break;
44.         case 2:output(head);break;
45.         case 3:change(&head, n);break;
46.         case 4:c_aver(head, n, &head1);break;
47.         case 5:print_all(head, head1);break;
48.         default :if(t)
49.             {
50.                 while(head)
51.                 {
52.                     stu *p = head->next;
53.                     free(head);
54.                     head = p;
55.                 }
56.                 free(head);
57.             }
58.         return 0;

```

```

59.     }
60. }
61.}
62.
63. void input(stu **headp)
64. {
65.     stu *loc_head, *tail;
66.     printf("number of students:\t");
67.     scanf("%d", &n);
68.     printf("\n");
69.     loc_head = (stu*)malloc(sizeof(stu));
70.     tail = loc_head;
71.     for(int i = 0; i < n - 1; i++)
72.     {
73.         tail->next = (stu*)malloc(sizeof(stu));
74.         tail = tail->next;
75.     }
76.     tail->next = NULL;
77.     //至此完成空链表的构建
78.     stu *p = loc_head;
79.     for(int i = 0; p != NULL; i++)
80.     {
81.         printf("student%d:\n", i);
82.         printf("student number:\t");
83.         scanf("%s", p->num);
84.         printf("student name:\t");
85.         scanf("%s", p->name);
86.         printf("english:\t");
87.         scanf("%d", &p->english);
88.         printf("math:\t\t");

```

```

89.     scanf("%d", &p->math);
90.     printf("physics:\t");
91.     scanf("%d", &p->physics);
92.     printf("C-programing:\t");
93.     scanf("%d", &p->c);
94.     printf("\n");
95.     p = p->next;
96. }
97. //录入链表数据域
98. *headp = loc_head;
99. printf("\n\tmission acomplished!\n\n");
100. }
101.
102. void output(stu *headp)
103. {
104.     stu *p = headp;
105.     for(int i = 0; p != NULL; i++)
106.     {
107.         printf("information of student%d:\n", i);
108.         printf("student number:\t\t\t%s\n", p->num);
109.         printf("student name:\t\t\t%s\n", p->name);
110.         printf("student english score:\t\t%d\n", p->english);
111.         printf("student math score:\t\t%d\n", p->math);
112.         printf("student physics score:\t\t%d\n", p->physics);
113.         printf("student C-programing score:\t%d\n", p->c);
114.         p = p->next;
115.         printf("\n");
116.     }
117.     printf("\n\tmission acomplished!\n\n");
118. }

```

```

119.
120. void change(stu **headp, int n)
121. {
122.     int tnode;
123.     printf("change the information of student:\t");
124.     scanf("%d", &tnode);
125.     while(tnode > n - 1)
126.     {
127.         printf("No such student!\n");
128.         scanf("%d", &tnode);
129.     }
130.     stu *p = *headp;
131.     for(int i = 0; i != tnode; i++)
132.         p = p->next;
133.     printf("student%d:\n", tnode);
134.     printf("student number:\t");
135.     scanf("%s", p->num);
136.     printf("student name:\t");
137.     scanf("%s", p->name);
138.     printf("english:\t");
139.     scanf("%d", &p->english);
140.     printf("math:\t\t");
141.     scanf("%d", &p->math);
142.     printf("physics:\t");
143.     scanf("%d", &p->physics);
144.     printf("C-programing:\t");
145.     scanf("%d", &p->c);
146.     printf("\n\tmission acomplished!\n\n");
147. }
148.

```

```

149. void c_aver(stu*headp, int n, aver **head1p)
150. {
151.     stu *p = headp;
152.     aver *loc_head = NULL, *tail, *p1;
153.     loc_head = (aver*)malloc(sizeof(aver));
154.     tail = loc_head;
155.     for(int i = 0; i < n - 1; i++)
156.     {
157.         tail->next = (aver*)malloc(sizeof(aver));
158.         tail = tail->next;
159.     }
160.     tail->next = NULL;
161.     p1 = loc_head;
162.     for(int i = 0; p; i++)
163.     {
164.         p1->s_aver = (double)(p->english + p->math + p->physics + p->c) /
            4;
165.         printf("average score of student%d:\t%.2f\n", i, p1->s_aver);
166.         p1 = p1->next;
167.         p = p->next;
168.     }
169.     *head1p = loc_head;
170.     printf("\n\tmission acomplished!\n\n");
171. }
172.
173. void print_all(stu *headp, aver *head1p)
174. {
175.     stu *p = headp;
176.     aver *p1 = head1p;
177.     for(int i = 0; p != NULL; i++)

```

```

178. {
179.     printf("all information of student%d:\n", i);
180.     printf("student number:\t\t\t%s\n", p->num);
181.     printf("student name:\t\t\t%s\n", p->name);
182.     printf("student english score:\t\t%d\n", p->english);
183.     printf("student math score:\t\t%d\n", p->math);
184.     printf("student physics score:\t\t%d\n", p->physics);
185.     printf("student C-programing score:\t%d\n", p->c);
186.     printf("student average score:\t\t%.2f\n", p1->s_aver);
187.     p = p->next;
188.     p1 = p1->next;
189.     printf("\n");
190. }
191. printf("\n\mission acomplished!\n\n");
192. }

```

```

1
number of students:      3

student0:
student number: 1
student name: 1
english: 1
math: 1
physics: 1
C-programing: 54

student1:
student number: 2
student name: 2
english: 2
math: 2
physics: 2
C-programing: 24

student2:
student number: 3
student name: 3
english: 3
math: 3
physics: 3
C-programing: 3

mission acomplished!

```

图 3.3-2-1 输入

```

2
information of student0:
student number:      1
student name:      1
student english score: 1
student math score: 1
student physics score: 1
student C-programing score: 54

information of student1:
student number:      2
student name:      2
student english score: 2
student math score: 2
student physics score: 2
student C-programing score: 24

information of student2:
student number:      3
student name:      3
student english score: 3
student math score: 3
student physics score: 3
student C-programing score: 3

mission acomplished!

```

图 3.3-2-2 不带均值输出


```

3
change the information of student:      4
No such student!
2
student2:
student number: 3
student name:   3
english:        3
math:           3
physics:        3
C-programing:   423

mission acomplished!

```

图 3.3-2-3 修改信息

```

4
average score of student0:      14.25
average score of student1:      7.50
average score of student2:      108.00

mission acomplished!

```

图 3.3-2-4 收集学生分数平均值

```

5
all information of student0:
student number:      1
student name:        1
student english score: 1
student math score:  1
student physics score: 1
student C-programing score: 54
student average score: 14.25

all information of student1:
student number:      2
student name:        2
student english score: 2
student math score:  2
student physics score: 2
student C-programing score: 24
student average score: 7.50

all information of student2:
student number:      3
student name:        3
student english score: 3
student math score:  3
student physics score: 3
student C-programing score: 423
student average score: 108.00

mission acomplished!

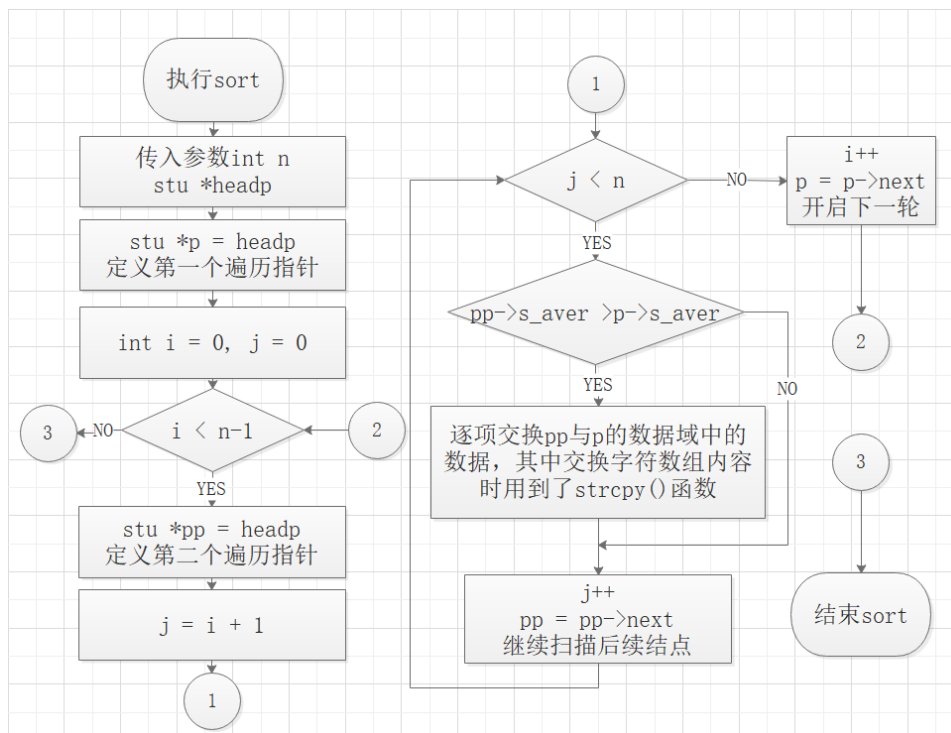
0
请按任意键继续. . .

```

图 3.3-2-5 输出学生所有信息&结束操作

本题要求加入根据学生平均分降序排列的操作，并且要求的是交换链表的数据域。

由于大量的代码片段以及输入输出结果的重复，在这里仅仅展示两个程序的差别处——选择排序交换数据域的流程图，代码，以及最后的结果展示。



```

1. void sort(stu *headp, int n)
2. {
3.     stu *p = headp;
4.     for(int i = 0; i < n - 1; i++, p = p->next)
5.     {
6.         stu *pp = p->next;
7.         for(int j = i + 1; j < n; j++, pp = pp->next)

```

```

8.      {
9.          if(pp->s_aver > p->s_aver)
10.         {
11.             double t = pp->s_aver;
12.             pp->s_aver = p->s_aver;
13.             p->s_aver = t;
14.             int e, m, ph, cp;
15.             e = pp->english;
16.             pp->english = p->english;
17.             p->english = e;
18.             m = pp->math;
19.             pp->math = p->math;
20.             p->math = m;
21.             ph = pp->physics;
22.             pp->physics = p->physics;
23.             p->physics = ph;
24.             cp = pp->c;
25.             pp->c = p->c;
26.             p->c = cp;
27.             char cnum[5];
28.             strcpy(cnum, pp->num);
29.             strcpy(pp->num, p->num);
30.             strcpy(p->num, cnum);
31.             char cname[10];
32.             strcpy(cname, pp->name);
33.             strcpy(pp->name, p->name);
34.             strcpy(p->name, cname);
35.         }
36.     }
37.

```

```

38. }
39. printf("\n\tmission acomplished!\n\n");
40.}

```

```

5
all information of student0:
student number:      1
student name:        1
student english score: 1
student math score:  1
student physics score: 23
student C-programing score: 34
student average score: 14.75

all information of student1:
student number:      2
student name:        2
student english score: 2
student math score:  242
student physics score: 432
student C-programing score: 423
student average score: 274.75

all information of student2:
student number:      3
student name:        3
student english score: 3
student math score:  34
student physics score: 2
student C-programing score: 4
student average score: 10.75

mission acomplished!

```

图 3.3-3-1 排序前

```

5
all information of student0:
student number:      2
student name:        2
student english score: 2
student math score:  242
student physics score: 432
student C-programing score: 423
student average score: 274.75

all information of student1:
student number:      1
student name:        1
student english score: 1
student math score:  1
student physics score: 23
student C-programing score: 34
student average score: 14.75

all information of student2:
student number:      3
student name:        3
student english score: 3
student math score:  34
student physics score: 2
student C-programing score: 4
student average score: 10.75

mission acomplished!

```

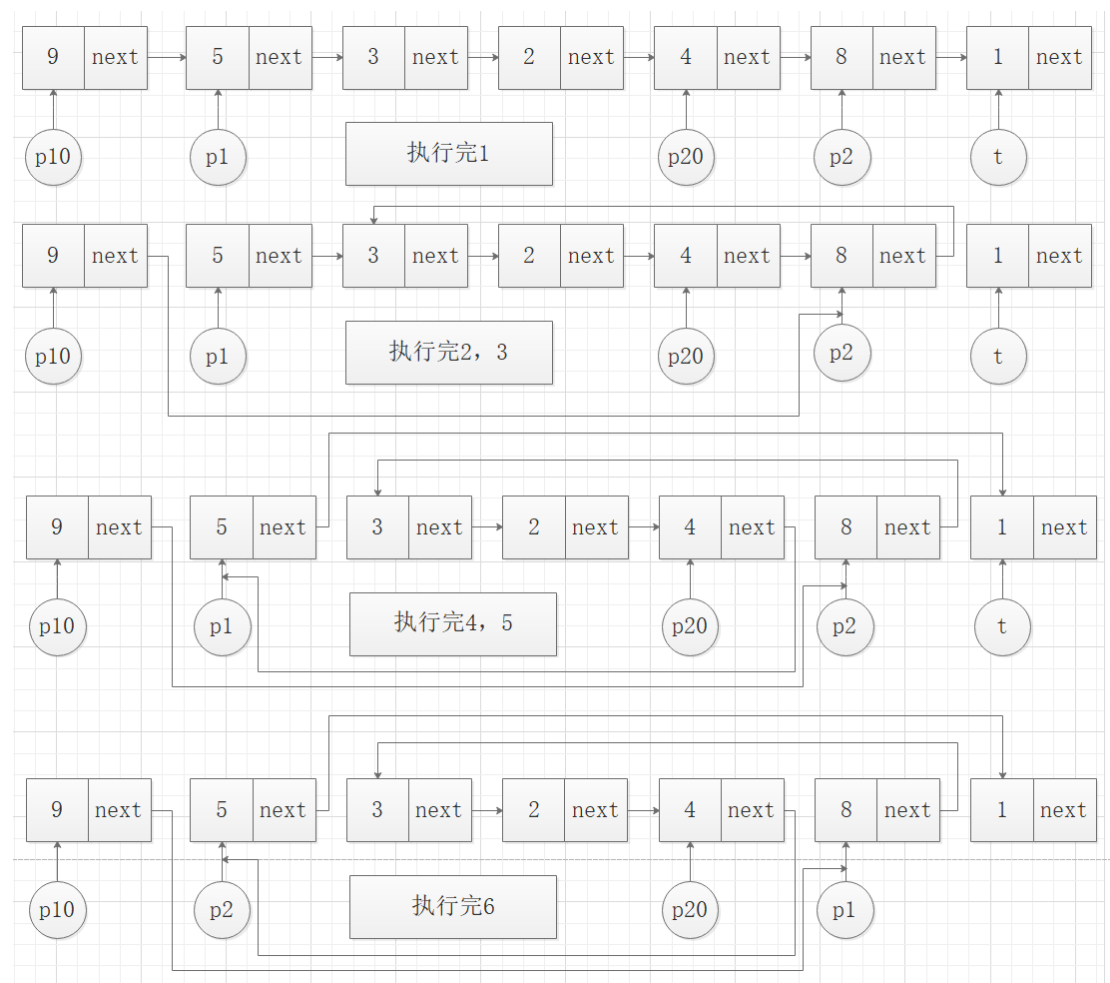
图 3.3-3-2 排序后

(4)

承接上题，这次改用直接交换结点的选择法。由于自己第一次编写的排序程序出现了逻辑错误，故在此首先给出我在自己编写排序之前的画图过程与思考。

第一次设想的交换结点的操作：

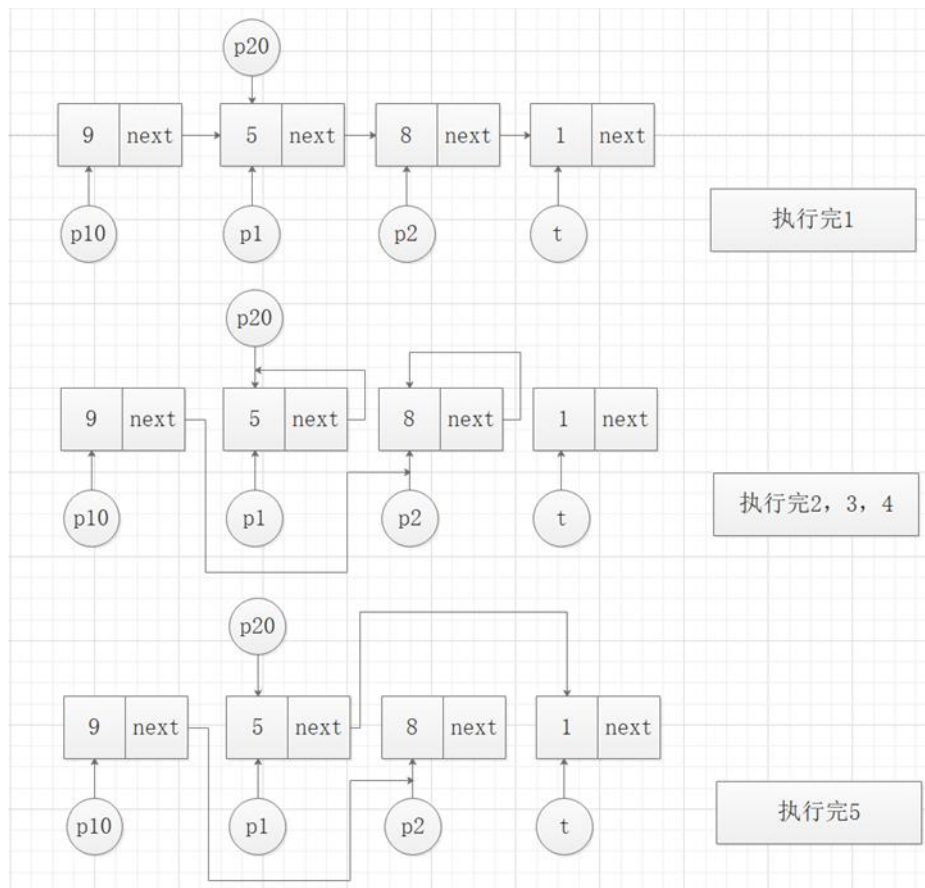
1. stu *t = p2->next;
2. p10->next = p2;
3. p2->next = p1->next;
4. p20->next = p1;
5. p1->next = t;
6. t = p1, p1 = p2, p2 = t;



看上去似乎比较完美地完成了所需的交换，但是在程序运行时出现了数据丢失的现象，就此推断应当是在交换结点时存在疏忽，导致在一个特殊情况下会导致链的断裂。

然后经过分析，最有可能出现问题的地方就在头部进行交换时，若 **p1** 所指向的结点与 **p20** 指向的结点相同情况之中。于是缩短了链的长度，针对头部交换情况再次作图分析如下。

由此可见在执行完了真正有效的交换操作（步骤 6 是为了最后使得 **p1**, **p2** 指向的仍是 **p10**, **p20** 指向的后继结点）之后确实会发生断链，因此虽然处理一般情况时，之前给出的算法步骤可行，然而并不符合所有情况。根据上课所学知识，推测与操作顺序有关。

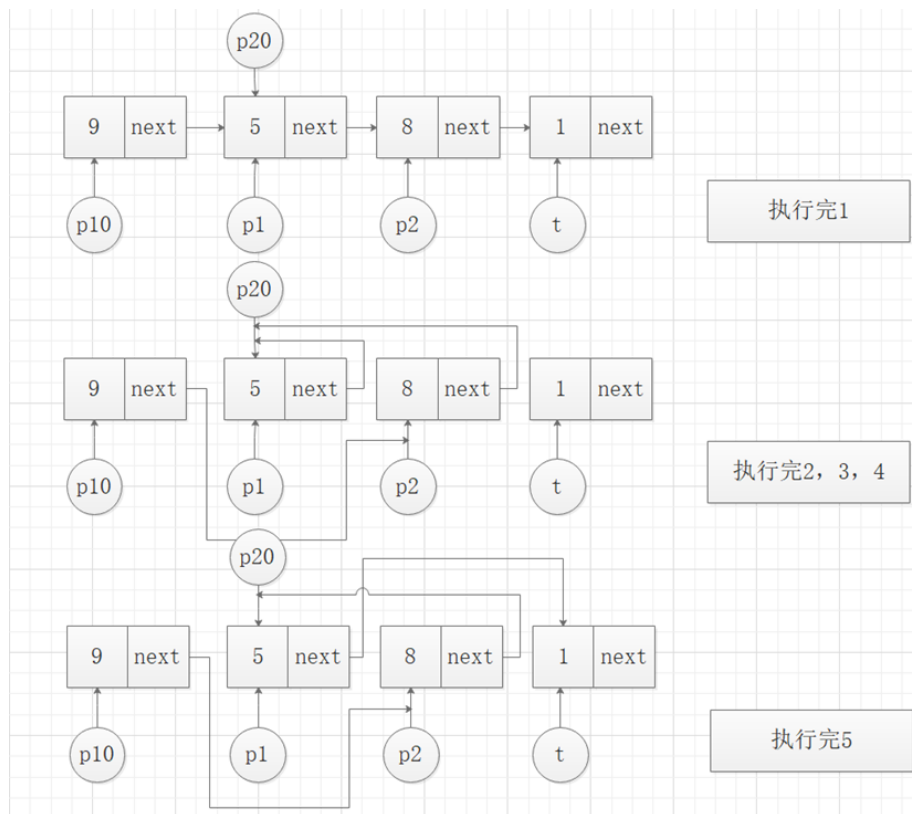


故考虑置换结点交换的顺序，经过画图的试探，以及程序代码的尝试之后修改成如下形式：

1. `stu *t = p2->next;`
2. `p10->next = p2;`
3. `p20->next = p1;`
4. `p2->next = p1->next;`
5. `p1->next = t;`
6. `t = p1;`
7. `p1 = p2;`
8. `p2 = t;`

再次绘制图像分析如下。

可以看出，这次的操作就没有导致断链现象的发生，因此是正确的操作顺序。基于这一认识，给出完整的 `sort` 函数的代码如下：



```

1. void sort(stu **headp)
2. {
3.     stu *p10 = (stu*)malloc(sizeof(stu));
4.     p10->next = *headp;
5.     *headp = p10;
6.     stu *p1, *p20, *p2;
7.     p1 = p10->next;
8.     for(; p1->next; p10 = p1, p1 = p1->next)
9.         for(p20 = p1, p2 = p20->next; p2; p20 = p2, p2 = p2->next)
10.            {
11.                if(p1->s_aver < p2->s_aver)
12.                    {
13.                        stu *t = p2->next;
14.                        p10->next = p2;
15.                        p20->next = p1;
16.                        p2->next = p1->next;

```

```

17.      p1->next = t;
18.      t = p1;
19.      p1 = p2;
20.      p2 = t;
21.  }
22.  }
23.  *headp = (*headp)->next;
24.  printf("\n\tmission acomplished!\n\n");
25.}

```

很直观的感受就是这次的代码远比上一次编写的交换数据域的 `sort` 函数简洁。这就说明在结构的数据域较为复杂的前提下，编写直接交换结点的程序虽然在思维层面上较为复杂，但是实现时的代码任务量较为轻松，且由于是直接以指针形式进行操作，效率同样由于前者。

```

5
all information of student0:
student number:      1
student name:        1
student english score: 1
student math score:  3
student physics score: 4
student C-programing score: 6
student average score: 3.50

all information of student1:
student number:      72
student name:        34
student english score: 345
student math score:  345
student physics score: 345
student C-programing score: 53
student average score: 272.00

all information of student2:
student number:      7
student name:        2345
student english score: 435
student math score:  1
student physics score: 1
student C-programing score: 3465
student average score: 975.50

mission acomplished!

```

图 3.3-4-1 排序前

```

5
all information of student0:
student number:      7
student name:        2345
student english score: 435
student math score:  1
student physics score: 1
student C-programing score: 3465
student average score: 975.50

all information of student1:
student number:      72
student name:        34
student english score: 345
student math score:  345
student physics score: 345
student C-programing score: 53
student average score: 272.00

all information of student2:
student number:      1
student name:        1
student english score: 1
student math score:  3
student physics score: 4
student C-programing score: 6
student average score: 3.50

mission acomplished!

```

图 3.3-4-2 排序后

(5)

要求使用单向链表来完成高精度计算。首先回忆一下之前讲解实现过的基于分治思想的最简单的高精度计算的方法，我们在做加法的时候是利用了一个通过动态申请空间大小的数组来储存数据的。而一个值得注意的细节则是，在储存时我们是类似于反序存储的，将数据的低位存在数组的低位之中。为什么要采用这种方法呢？如果仅仅是因为这么读取内容符合数组的结构，不需要读入之后再做调整，那么在用链表实现时究竟应当使用后进先出链表还是先进先出链表呢？

仔细分析，“倒序”存储不仅仅是为了方便存储，更是为了方便计算。因为我们的加法是从低位向高位进行的，以适应进位的机制。因此我们再通过链表实现算数加法的模拟时应当建立两个先进先出的链表用来处理加数和被加数。

那么应当用什么存储最终结果呢？最为方便的当然是同样构建一个后进先出的链表。这样就可以将求得的和的数据从低位向高位存储，然后从高位向低位输出结果就可以了。

但是我在完成实验时正好在学习有关链表的一些常规操作，自己构建了一个反转单向链表的算法。因此我们构建一个足够大的先进先出的链表来存储加和，然后将其倒置再输出。现在下方内容给出执行反转操作的代码以及代码执行的操作的过程的图示。

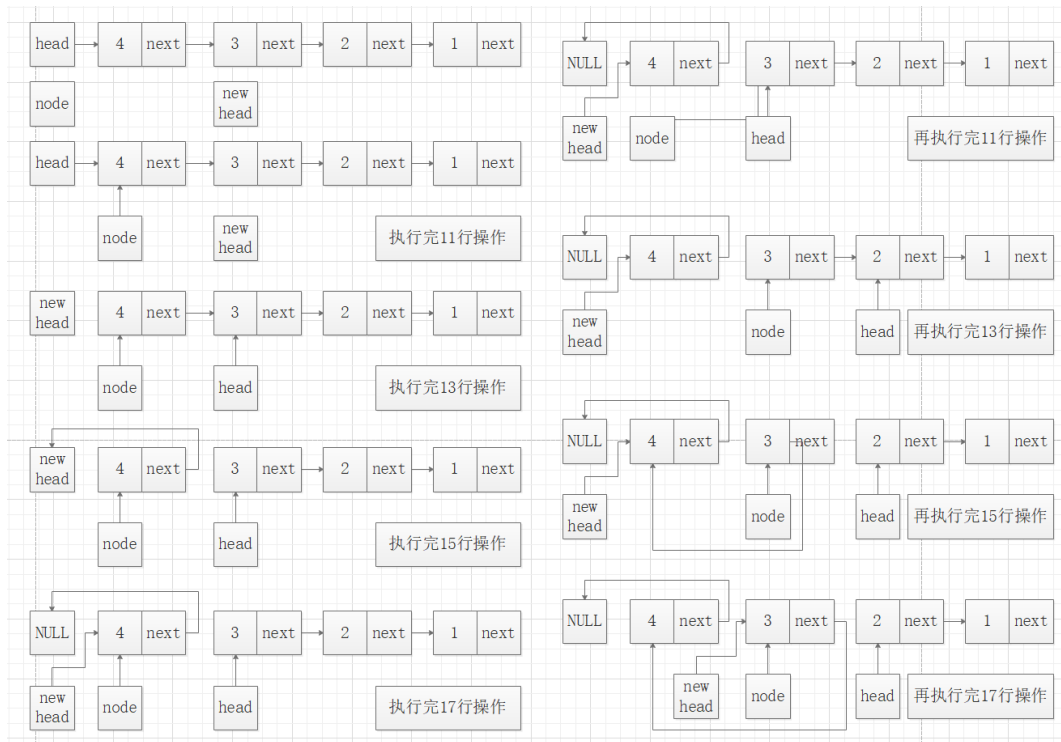
反转的函数代码如下：

```
1. void reverse(list **ansp)
2. {
3.     list *head = *ansp;
4.     //head 为原链表的头指针
5.     list *newhead = NULL;
6.     //newhead 为反转后链表的头结点指针
7.     list *node = NULL;
8.     //node 为中间操作结点
9.     while(head)
10.    {
11.        node = head;
```

```

12. //使 node 指向当前的头结点
13. head = head->next;
14. //相当于做了对原链表执行了“头删除”的操作
15. node->next = newhead;
16. //使切掉的结点指向新链头结点
17. newhead = node;
18. //更新新链的头结点
19. }
20. *ansp = newhead;
21.}

```



可以看出，在经过了两轮的操作之后，成功地构建了从 3 到 4 的新链，继续操作就将完成整个链表的反转。

接下来是程序其余部分的代码以及操作的结果。

```

1. #include <stdio.h>
2. #include <stdlib.h>
3.

```

```

4.  typedef struct list{
5.      int data;
6.      struct list *next;
7.  }list;
8.
9.  void create(list **headp, int *n);
10. void add(list *headp1, list *headp2, list **ansp, int n);
11. void reverse(list **ansp);
12.
13. int n1, n2, *np1 = &n1, *np2 = &n2;
14.
15. int main()
16. {
17.     list *head1 = NULL, *head2 = NULL, *ans = NULL;
18.     head1 = (list*)malloc(sizeof(list));
19.     head2 = (list*)malloc(sizeof(list));
20.     create(&head1, np1);
21.     create(&head2, np2);
22.     int n = (n1>n2)?(n1+1):(n2+1);
23.     ans = (list*)malloc(sizeof(list));
24.     add(head1, head2, &ans, n);
25.     reverse(&ans);
26.     if(ans->data)
27.         printf("%d", ans->data);
28.     list *p = ans->next;
29.     while(p)
30.     {
31.         printf("%d", p->data);
32.         p = p->next;
33.     }

```

```

34. printf("\n");
35. while(ans)
36. {
37.     p = ans->next;
38.     free(ans);
39.     ans = p;
40. }
41. free(ans);
42. while(head1)
43. {
44.     p = head1->next;
45.     free(head1);
46.     head1 = p;
47. }
48. free(head1);
49. while(head2)
50. {
51.     p = head2->next;
52.     free(head2);
53.     head2 = p;
54. }
55. free(head2);
56. return 0;
57. }
58.
59. void create(list **headp, int *n)
60. {
61.     *n = 0;
62.     list *loc_head = NULL;
63.     char c;

```

```

64. loc_head = (list*)malloc(sizeof(list));
65. loc_head->data = -1;
66. loc_head->next = NULL;
67. while((c = getchar()) != '\n')
68. {
69.     list *cur = (list *)malloc(sizeof(list));
70.     cur->data = c - '0';
71.     cur->next = loc_head;
72.     loc_head = cur;
73.     (*n)++;
74. }
75. //构建并录入了先进后出的链表,使用先进后出链表是基于方便加法进
    位的考虑
76. *headp = loc_head;
77.}
78.
79. void add(list *headp1, list *headp2, list **ansp, int n)
80. {
81.     list *loc_ans = NULL, *tail;
82.     loc_ans = (list*)malloc(sizeof(list));
83.     loc_ans->next = NULL;
84.     tail = loc_ans;
85.     for(int i = 0; i < n; i++)
86.     {
87.         tail->next = (list*)malloc(sizeof(list));
88.         tail = tail->next;
89.     }
90.     tail->next = NULL;
91. //构建足够大(刚好或者大一个结点)的先进先出链表
92. int carry = 0;

```

```

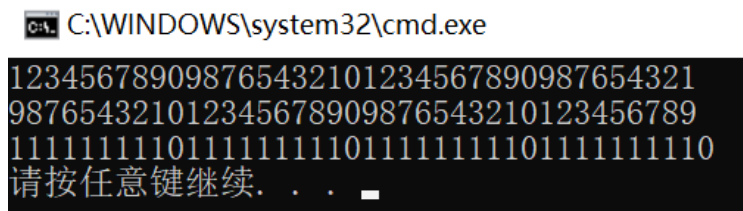
93. list *h1 = headp1, *h2 = headp2;
94. list *p = loc_ans;
95. while(h1->data != -1 && h2->data != -1)
96. {
97.     p->data = h1->data + h2->data + carry;
98.     carry = p->data / 10;
99.     p->data %= 10;
100.    p = p->next;
101.    h1 = h1->next;
102.    h2 = h2->next;
103. }
104. if(h1->data != -1)
105. {
106.     while(h1->data != -1)
107.     {
108.         p->data = h1->data + carry;
109.         carry = p->data / 10;
110.         p->data %= 10;
111.         p = p->next;
112.         h1 = h1->next;
113.     }
114.     p->data = carry;
115. }
116. else if(h2->data != -1)
117. {
118.     while(h2->data != -1)
119.     {
120.         p->data = h2->data + carry;
121.         carry = p->data / 10;
122.         p->data %= 10;

```

```

123.     p = p->next;
124.     h2 = h2->next;
125. }
126.     p->data = carry;
127. }
128. else
129. {
130.     p->data = carry;
131. }
132.     p->next = NULL;
133.     *ansp = loc_ans;
134. }

```



```

C:\WINDOWS\system32\cmd.exe
123456789098765432101234567890987654321
987654321012345678909876543210123456789
111111110111111110111111110111111110
请按任意键继续. . .

```

图 3.3-5 链表实现高精度计算

(6)

本题要求的是输入已经给定的逆波兰表达式，然后进行求值。首先考虑一下如何将中缀表达式转换为逆波兰表达式。

应当构建两个栈，一个 S1 先用来存储运算符（最后为空），另一个 S2 存储最终的结果。扫描中缀表达式，若为数值，将它压入栈 S2。若扫描到的为左括号 “[”，则将其直接入栈 S1；若为右括号 “]”，则将左右括号之间的所有运算符依次出栈并压入栈 S2。若为运算符，则按下列操作判断：

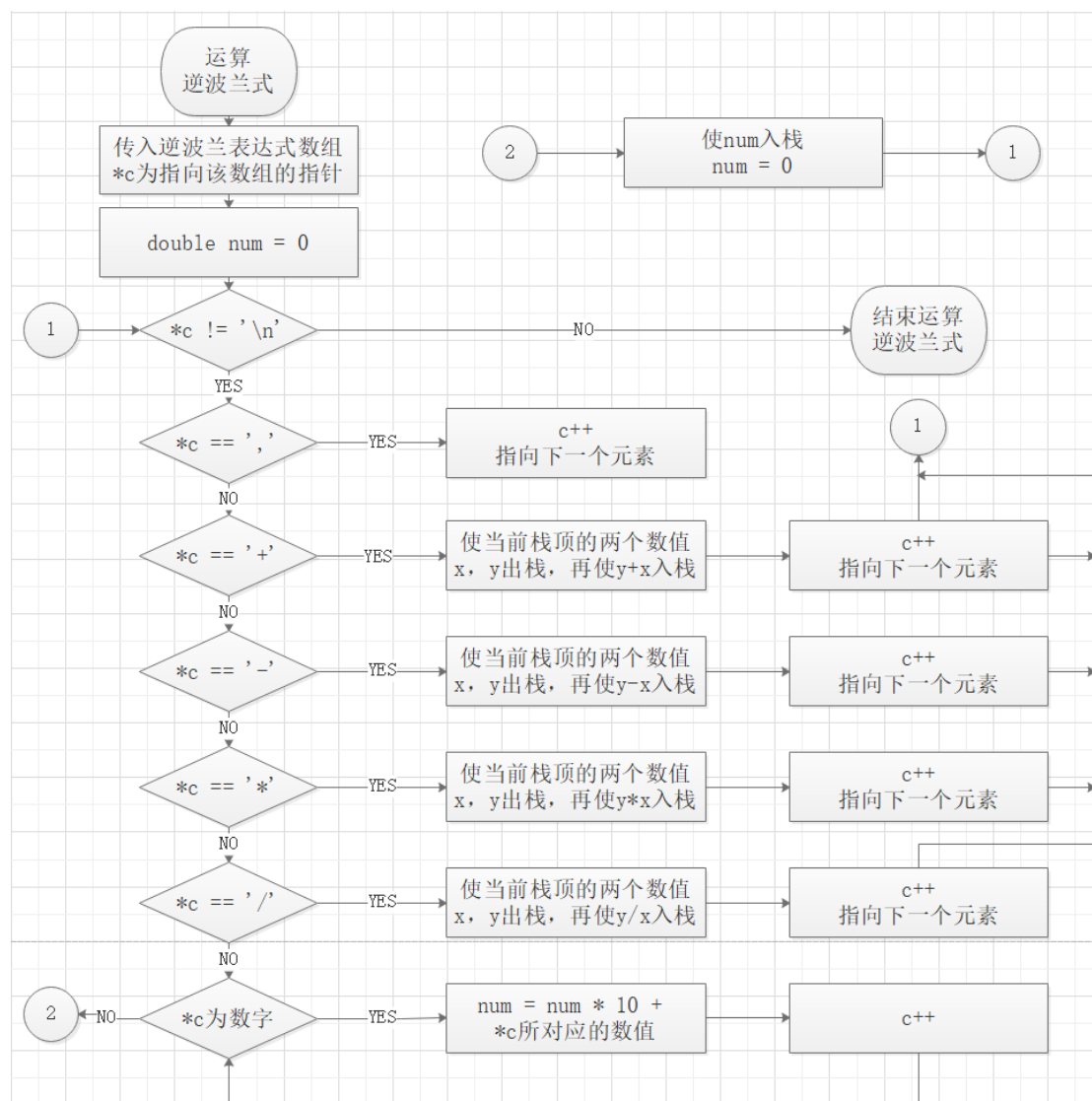
- 1) 若 S1 栈为空，或栈顶为左括号 “[” 则直接入栈
- 2) 若 S1 栈非空，且栈顶为运算符，则判断如下

若栈顶运算符优先级不低于当前运算符，则将栈顶运算符出栈并重复当

前判断直至栈顶运算符优先级低于当前运算符，将当前运算符压栈

如果最后对中缀表达式扫描结束了，则将运算符栈 S1 中的所有运算符依次出栈并压栈进入 S2。至此就可以在 S2 栈中得到一个中缀表达式转化而来的逆波兰式了。

然后就是运算一个逆波兰表达式的值了。算法流程图如下：



程序实现的代码如下。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. typedef struct node{
5.     double data;
```



```

6.  struct node *next;
7. }node;
8.
9.  typedef struct stack{
10.   node *st;
11. }stack;
12.
13. void create(stack*);
14. void check(stack*, char *);
15. void push(stack*, double);
16. double pop(stack*);
17.
18. int main()
19. {
20.   stack stack0;
21.   create(&stack0);
22.   char c[100];
23.   fgets(c, 100, stdin);
24.   check(&stack0, c);
25.   printf("%f\n", pop(&stack0));
26.   free((&stack0)->st);
27.   //在 pop 时就会 free 掉原本的栈顶空间,最后只用 free 掉&stack0->st
28.   return 0;
29. }
30.
31. void create(stack *stp)
32. {
33.   node *cur;
34.   cur = (node*)malloc(sizeof(node));
35.   cur->next = NULL;

```

```

36.  stp->st = cur;
37. }
38.
39. void check(stack *stp, char *c)
40. {
41.     double num = 0;
42.     while(*c != '\n')
43.     {
44.         switch(*c)
45.         {
46.             case '+':break;
47.             case '+':double x1, y1;
48.                 x1 = pop(stp);
49.                 y1 = pop(stp);
50.                 push(stp, y1 + x1);
51.             break;
52.             case '-':double x2, y2;
53.                 x2 = pop(stp);
54.                 y2 = pop(stp);
55.                 push(stp, y2 - x2);
56.             break;
57.             case '*':double x3, y3;
58.                 x3 = pop(stp);
59.                 y3= pop(stp);
60.                 push(stp, y3 * x3);
61.             break;
62.             case '/':double x4, y4;
63.                 x4 = pop(stp);
64.                 y4 = pop(stp);
65.                 push(stp, y4 / x4);

```

```

66.         break;
67.         default:while(*c >= '0' && *c <= '9')
68.         {
69.             num = num * 10 + *c - '0';
70.             c++;
71.         }
72.         c--;
73.         push(stp, num);
74.         num = 0;
75.     }
76.     c++;
77. }
78.}
79.
80. void push(stack *stp, double p)
81. {
82.     node *cur;
83.     cur = (node*)malloc(sizeof(node));
84.     stp->st->data = p;
85.     cur->next = stp->st;
86.     stp->st = cur;
87. }
88.
89. double pop(stack *stp)
90. {
91.     node *cur = stp->st;
92.     stp->st = cur->next;
93.     free(cur);
94.     return stp->st->data;
95. }

```

表 3.3-6 逆波兰式求值

序号	中缀表达式与结果	逆波兰式	计算正误
1	$(1*2+3)*4+5 == 25$	1,2,*,3,+,4,*,5,+	√
2	$1+2-3+4-5 == -1$	1,2,+,3,-,4,+,5,-	√
3	$1/10+2*5 == 10.10$	1,10,/,2,5,*,+	√

```
C:\WINDOWS\system32\cmd.exe
1, 2, *, 3, +, 4, *, 5, +
25.000000
请按任意键继续. . .
```

图 3.3-6-1 序号 1

```
C:\WINDOWS\system32\cmd.exe
1, 2, +, 3, -, 4, +, 5, -
-1.000000
请按任意键继续. . .
```

图 3.3-6-2 序号 2

```
C:\WINDOWS\system32\cmd.exe
1, 10, /, 2, 5, *, +
10.100000
请按任意键继续. . .
```

图 3.3-6-3 序号 3

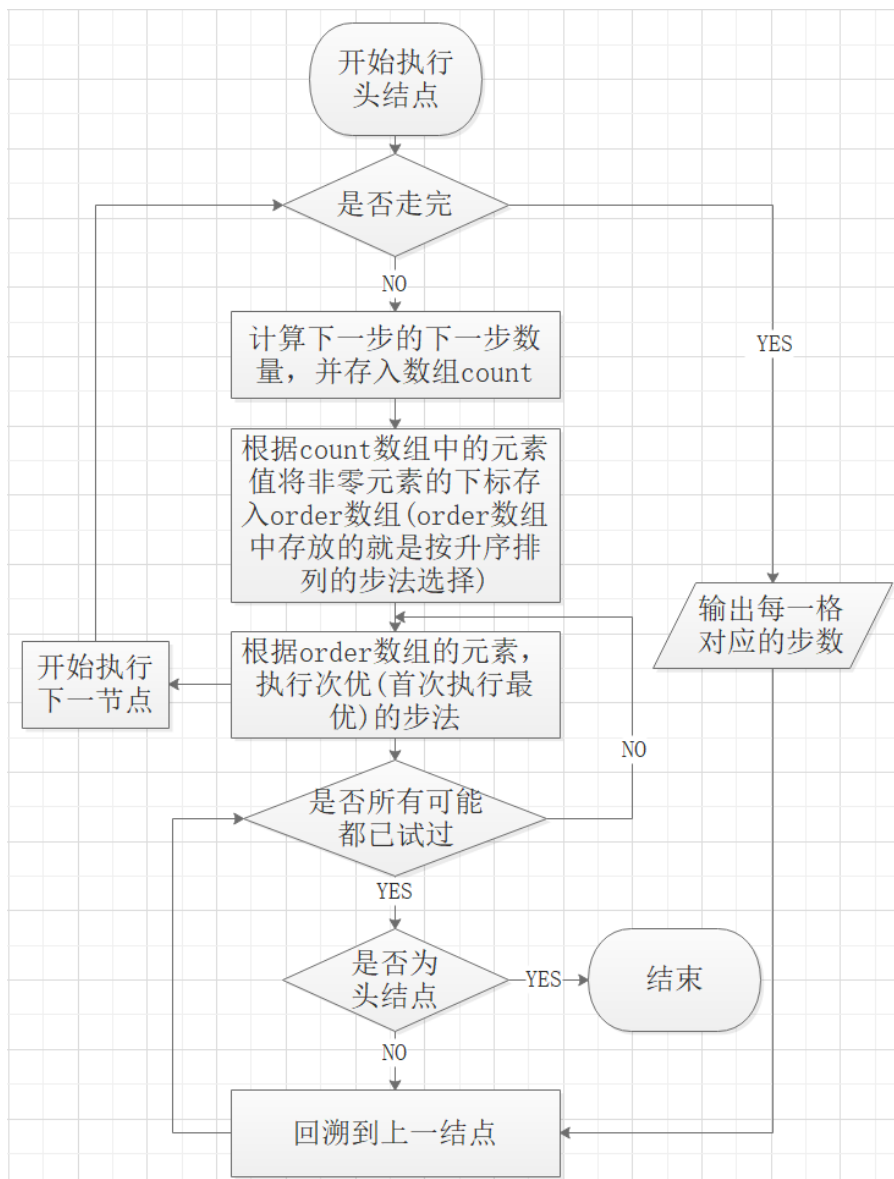
(7)

马踏棋盘问题，在我看来与上一节中的迷宫问题其实有些类似，都有确定的走法以及相似的限制条件。然而这一问题更为复杂的它的路数过多，如果是一个 64 位的棋盘，那么如果在走完前 10 步之后其实就已经明确了继续这么走下去是没有解的，但是很有可能直到最后几步，都是存在下一步的解空间的，那么就会不断地往后执行。而最终碰壁之后，我们就要一个结点一个结点地往前回溯，直到能够回到第 10 步。这样程序的执行效率就会很低。但是同时我们也能看出，如果想要完全找出所有的解，唯一的方法就是从第一步开始，考虑每一种解的可能，也就是必须便利所有 8^{64} 种可能。而这每一种可能均有可能会遇到前面提

到的结点过深，回溯过于困难的问题。因此需要有选择地执行下一步操作。

考虑到在计算思维课程之中了解到的贪心算法，我们最直接的选择就是每一步都选择最佳的方案。那么什么是最佳的方案呢？由上所述，可知困难的关键在于回溯。每一轮的回溯都会将那一支的所有可能尝试完毕之后继续回溯，所以最好使的最开始的回溯尽可能的少，那么我们在执行时就应当优先选择下下一步尽可能少的下一步。也就是说要按照下下一步数目的升序排列依次选择可行的下一步。

主体的流程图如下：



实现代码：

```
1. #include <stdio.h>
```

```

2. #include <stdlib.h>
3.
4. #define row 15
5. #define col 15
6. #define maxstep(x, y) (x * y)
7.
8. typedef struct{
9.     int x;
10.    int y;
11.    int direction;
12. }stack;
13. stack step[maxstep(row, col)];
14. //结构数组实现栈
15.
16. int chess[row+1][col+1]={0};
17. int cx[8]={1, -1, -2, 2, 2, 1, -1, -2};
18. int cy[8]={2, -2, 1, 1, -1, -2, 2, -1};
19. int top = -1;
20. int sol = 0; //需要的解的个数
21. int opt = 0;
22. int soln = 0; //已运行的解的个数
23. int Max = maxstep(row, col);
24.
25. void run();
26. void print();
27. void push(int, int);
28. void pop();
29. int check(int, int);
30.
31. int main(void)

```

```

32. {
33.     int x, y;
34.     printf("enter original position:");
35.     scanf("%d%d", &x, &y);
36.     printf("\t1\tprint one route\n");
37.     printf("\t2\tprint all routes\n");
38.     printf("\t3\tprint certain numbers of routes\n");
39.     scanf("%d", &opt);
40.     if(opt == 3)
41.     {
42.         printf("numbers of routes you need:\t");
43.         scanf("%d",&sol);
44.         opt = 2;
45.     }
46.     push(x, y);
47.     chess[x][y] = top + 1;
48.     run();
49. }
50.
51. void run()
52. {
53.     int x0, y0;
54.     while(1) {
55.         if(opt == 1)
56.         {
57.             if(top == Max - 1)
58.             {
59.                 print();
60.                 break;
61.             }

```

```

62.     }
63.     else if(opt == 2 )
64.     {
65.         if(top == Max - 1)
66.         {
67.             soln++;
68.             printf("%d \n\n", soln);
69.             print();
70.         }
71.     }
72.     if(sol != 0 && soln == sol)
73.         break;
74.     //以上为输出结果段
75.     x0 = step[top].x;
76.     y0 = step[top].y;
77.     int count[8]={0};
78.     for(int i = 0;i < 8;i++)
79.     {
80.         int x1 = x0 + cx[i];
81.         int y1 = y0 + cy[i];
82.         if(check(x1,y1))
83.         {
84.             for(int j = 0;j < 8;j++)
85.             {
86.                 int x2 = x1 + cx[j];
87.                 int y2 = y1 + cy[j];
88.                 if(check(x2,y2))
89.                     count[i]++;
90.                 //计算下一步的下一步有多少种可能并存入数组 count
91.             }

```



```

92.     }
93. }
94. int order[8] = {0};
95. int temp = 9;
96. int k = 0;
97. for(int i = 0; i < 8; i++)
98. {
99.     temp = 9;
100.     for(int j = 0; j < 8; j++)
101.     {
102.         if(count[j] < temp && count[j])
103.         {
104.             order[i] = j;
105.             temp = count[j];
106.             k = j;
107.         }
108.     }
109.     count[k] = 0;
110. }
111. //完成路径数的升序排列,“舍弃”count 数组内为 0 的元素
112. int direnow = 0;
113. for(direnow = step[top].direction + 1 ; direnow < 8 ; direnow++)
114. {
115.     int xNext = x0, yNext = y0;
116.     xNext += cx[order[direnow]];
117.     yNext += cy[order[direnow]];
118.     step[top].direction += 1;
119.     if(check(xNext, yNext))
120.     {
121.         push(xNext, yNext);

```

```

122.     chess[xNext][yNext] = top + 1;
123.     break;
124. }
125. }
126. //优先按照下下一步少的下一步进行
127. if(step[top].direction == 7)
128. {
129.     int x, y;
130.     x = step[top].x;
131.     y = step[top].y;
132.     chess[x][y] = 0;
133.     pop();
134. }
135. //回溯
136. }
137. }
138.
139. void push(int xx, int yy)
140. {
141.     ++top;
142.     step[top].x = xx;
143.     step[top].y = yy;
144.     step[top].direction = -1;
145. }
146.
147. void pop()
148. {
149.     step[top].x = 0;
150.     step[top].y = 0;
151.     step[top].direction = -1;

```

```

152.  --top;
153. }
154.
155. int check(int xx,int yy)
156. {
157.     if(xx > 0 && xx <= row && yy > 0 && yy <= col && chess[xx][yy] == 0
        )
158.         return 1;
159.     else return 0;
160. }
161.
162. void print()
163. {
164.     printf("route:\n");
165.     for(int i = 1;i <= row;i++)
166.     {
167.         for(int j = 1;j <= col;j++)
168.             printf("%6d ", chess[i][j]);
169.         printf("\n");
170.     }
171.     printf("\n\n");
172. }

```

说明：这里有三种模式，1 对应输出找到的第一组解，2 对应输出“所有解”，这是需要很长很长的时间的，3 对应输出特定数目的解。

```
C:\WINDOWS\system32\cmd.exe
enter original position:
2 3
    1      print one route
    2      print all routes
    3      print certain numbers of routes
3
numbers of routes you need:    2
1
route:
    2      31      4      19      24      29      14      17
    5      20      1      30      15      18      25      28
   32      3      56      23      48      27      16      13
   21      6      33      46      57      40      49      26
   34      55      22      63      44      47      12      39
    7      62      45      58      37      64      41      50
   54      35      60      9      52      43      38      11
   61      8      53      36      59      10      51      42

2
route:
    2      31      4      19      24      29      14      17
    5      20      1      30      15      18      25      28
   32      3      56      23      48      27      16      13
   21      6      33      46      57      40      49      26
   34      55      22      63      44      47      12      39
    7      62      45      58      37      64      41      50
   54      35      60      9      52      43      38      11
   61      8      53      36      59      10      51      42

请按任意键继续. . .
```

图 3.3-7 马踏棋盘问题

3.4 小结

在本节实验中首先是通过表达式求值的程序验证巩固了有关结构与指针的概念，回顾了各类运算符的优先级关系。

然后通过程序修改体会了构建先进先出链表的方法，以及传入结构指针变量的指针的原因——如果要在函数内修改一个变量的值并使其作用到函数体之外，就需要传入待改变的量的指针进行操作。而后的建立后进先出链表则是学习了一种“新的”数据存储的类型。在后续的实验中也体现出了其价值。

第 1 道实验题主要是用来熟悉字段结构以及联合的使用方法，其中也运用到了如构建函数指针数组的用法，理解了字段结构与联合的空间存储与分配的方式，

并加以运用。

第 2, 3, 7 道实验题, 则是对单向链表结构的逐层深入的学习。首先通过第 2 题巩固了链表的输入, 输出, 插入, 修改等操作。然后第 5 题以交换数据域的方法实现了链表的重排, 也感受到了这种方法在结构数据域内的数据之类复杂的情况下的代码书写量的繁重。所以紧接着的第 7 题就改用交换结点的选择排序法改进了程序。在发现了运行错误之后, 分析选择排序的算法执行时通过绘制运行程序图示的方法发现并更正了原本设想中的错误, 加深了对交换结点时的顺序的重要性的认识。

第 5 题通过链表实现了之前用数组实现的高精度加法计算。通过分析选择了合适的后进先出链表。然后又为了将自己思考的链表倒置的算法加入进去而将最终结果的链表类型改为先进先出, 然后通过头结点逐次更新的方式逆转了该链表, 达到了构建“后进先出”链表的目的。

第 9 题首先是学习了由中缀式转换为逆波兰式的方法以及逆波兰式的计算体系。然后通过用后进先出链表的方式构建了栈的结构, 构建了压栈, 出栈等函数以进行对应的操作。

最后的第 10 题, 在迷宫问题的基础之上分析了马踏棋盘问题与之的相似性与实现困难程度的差异性。在考虑到回溯次数的问题时, 利用计算思维课程中的贪心算法的思想对深度优先搜索加以改进, 降低了求解较小数目的解时的回溯次数进而提高了这一情形下的算法执行效率(求解所有解必须将所有可能的步法全部实验, 基本无法实现改进)。值得一提的是, 这一题的栈没有再使用链表构建, 而是通过结构数组实现的, 因为本题的栈的深度在限定完棋盘的大小之后就唯一确定了, 无需通过动态申请的方式就能够实现无冗余的存储, 而数组通过下标实现的“天然的”顺序结构也能很好的加以应用。

参考文献

- [1] 曹计昌, 卢萍, 李开. C 语言程序设计, 北京: 科学出版社, 2013
- [2] 卢萍, 李开, 王多强, 甘早斌. C 语言程序设计典型题解与实验指导, 北京: 清华大学出版社, 2019