1.汉若塔

说明河内之塔(Towers of Hanoi)是法国人M.Claus(Lucas)于1883年从泰国带至法国的，河内为越战时北越的首都，即现在的胡志明市；1883年法国数学家 Edouard Lucas曾提及这个故事，据说创世纪时Benares有一座波罗教塔，是由三支钻石棒（Pag）所支撑，开始时神在第一根棒上放置64个由上至下依由小至大排列的金盘（Disc），并命令僧侣将所有的金盘从第一根石棒移至第三根石棒，且搬运过程中遵守大盘子在小盘子之下的原则，若每日仅搬一个盘子，则当盘子全数搬运完毕之时，此塔将毁损，而也就是世界末日来临之时。

解法如果柱子标为ABC，要由A搬至C，在只有一个盘子时，就将它直接搬至C，当有两个盘子，就将B当作辅助柱。如果盘数超过2个，将第三个以下的盘子遮起来，就很简单了，每次处理两个盘子，也就是：A->B、A ->C、B->C这三个步骤，而被遮住的部份，其实就是进入程式的递回处理。事实上，若有n个盘子，则移动完毕所需之次数为2^n - 1，所以当盘数为64时，则所需次数为：2^{64}- 1 = 18446744073709551615为5.05390248594782e+16年，也就是约5000世纪，如果对这数字没什么概念，就假设每秒钟搬一个盘子好了，也要约5850亿年左右。

```
#include <stdio.h>

void hanoi(int n, char A, char B, char C) {
	if(n == 1) {
		printf("Move sheet %d from %c to %c\n", n, A, C);
	}
	else {
		hanoi(n-1, A, C, B);
		printf("Move sheet %d from %c to %c\n", n, A, C);
		hanoi(n-1, B, A, C);
	}
}

int main() {
	int n;
	printf("请输入盘数：");
	scanf("%d", &n);
	hanoi(n, 'A', 'B', 'C');
	return 0;
}
```

2.费式数列

说明

Fibonacci为1200年代的欧洲数学家，在他的着作中曾经提到：「若有一只兔子每个月生一只小兔子，一个月后小兔子也开始生产。起初只有一只兔子，一个月后就有两只兔子，二个月后有三只兔子，三个月后有五只兔子（小兔子投入生产）.....。
如果不太理解这个例子的话，举个图就知道了，注意新生的小兔子需一个月成长期才会投入生产，类似的道理也可以用于植物的生长，这就是Fibonacci数列，一般习惯称之为费氏数列，例如以下： 1、1 、2、3、5、8、13、21、34、55、89.....

解法

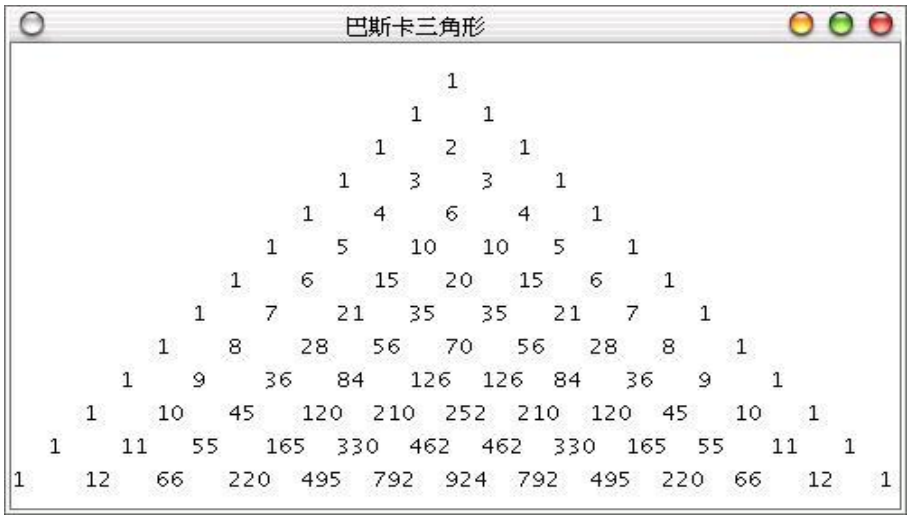
依说明，我们可以将费氏数列定义为以下：

fn = fn-1 + fn-2　　if n > 1

fn = n　　if n = 0, 1

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

#define N 20

int main(void) {
	int Fib[N] = {0};
	int i;

	Fib[0] = 0;
	Fib[1] = 1;

	for(i = 2; i < N; i++)
	Fib[i] = Fib[i-1] + Fib[i-2];

	for(i = 0; i < N; i++)
	printf("%d ", Fib[i]);
	printf("\n");

	return 0;
}
```

3. 巴斯卡三角形




```
#include <stdio.h>
#define N 12
long combi(int n, int r){
	int i;
	long p = 1;
	for(i = 1; i <= r; i++)
```

```
        p = p * (n-i+1) / i;
    return p;
}
void paint() {
    int n, r, t;
    for(n = 0; n <= N; n++) {
        for(r = 0; r <= n; r++) {
            int i;/* 排版设定开始 */
            if(r == 0) {
                for(i = 0; i <= (N-n); i++)
                    printf("  ");
            }else {
                printf("  ");
            } /* 排版设定结束 */
            printf("%3d", combi(n, r));
        }
        printf("\n");
    }
}
```

4.三色棋

说明

三色旗的问题最早由E.W.Dijkstra所提出，他所使用的用语为Dutch Nation Flag(Dijkstra为荷兰人)，而多数的作者则使用Three-Color Flag来称之。

假设有一条绳子，上面有红、白、蓝三种颜色的旗子，起初绳子上的旗子颜色并没有顺序，您希望将之分类，并排列为蓝、白、红的顺序，要如何移动次数才会最少，注意您只能在绳子上进行这个动作，而且一次只能调换两个旗子。

解法

在一条绳子上移动，在程式中也就意味只能使用一个阵列，而不使用其它的阵列来作辅助，问题的解法很简单，您可以自己想像一下在移动旗子，从绳子开头进行，遇到蓝色往前移，遇到白色留在中间，遇到红色往后移，如下所示：

只是要让移动次数最少的话，就要有些技巧：

如果图中W所在的位置为白色，则W+1，表示未处理的部份移至至白色群组。

如果W部份为蓝色，则B与W的元素对调，而B与W必须各+1，表示两个群组都多了一个元素。

如果W所在的位置是红色，则将W与R交换，但R要减1，表示未处理的部份减1。

注意B、W、R并不是三色旗的个数，它们只是一个移动的指标；什么时候移动结束呢？一开始时未处理的R指标会是等于旗子的总数，当R的索引数减至少于W的索引数时，表示接下来的旗子就都是红色了，此时就可以结束移动，如下所示：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BLUE 'b'
#define WHITE 'w'
```

```
#define RED 'r'

#define SWAP(x, y) { char temp; \
                     temp = color[x]; \
                     color[x] = color[y]; \
                     color[y] = temp; }

int main() {
    char color[] = {'r', 'w', 'b', 'w', 'w',
                    'b', 'r', 'b', 'w', 'r', '\0'};

    int wFlag = 0;
    int bFlag = 0;
    int rFlag = strlen(color) - 1;
    int i;

    for(i = 0; i < strlen(color); i++)
        printf("%c ", color[i]);
    printf("\n");

    while(wFlag <= rFlag) {
        if(color[wFlag] == WHITE)
            wFlag++;
        else if(color[wFlag] == BLUE) {
            SWAP(bFlag, wFlag);
            bFlag++; wFlag++;
        }
        else {
            while(wFlag < rFlag && color[rFlag] == RED)
               rFlag--;
            SWAP(rFlag, wFlag);
            rFlag--;
        }
    }

    for(i = 0; i < strlen(color); i++)
        printf("%c ", color[i]);
    printf("\n");

    return 0;
}
```

5.老鼠走迷官（一）

说明老鼠走迷宫是递回求解的基本题型，我们在二维阵列中使用2表示迷宫墙壁，使用1来表示老鼠的行走路径，试以程式求出由入口至出口的路径。

解法老鼠的走法有上、左、下、右四个方向，在每前进一格之后就选一个方向前进，无法前进时退回选择下一个可前进方向，如此在阵列中依序测试四个方向，直到走到出口为止，这是递回的基本题，请直接看程式应就可以理解。

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

int visit(int, int);

int maze[7][7] = {{2, 2, 2, 2, 2, 2, 2},
                  {2, 0, 0, 0, 0, 0, 2},
                  {2, 0, 2, 0, 2, 0, 2},
                  {2, 0, 0, 2, 0, 2, 2},
                  {2, 2, 0, 2, 0, 2, 2},
                  {2, 0, 0, 0, 0, 0, 2},
                  {2, 2, 2, 2, 2, 2, 2}};

int startI = 1, startJ = 1;   // 入口
int endI = 5, endJ = 5;   // 出口
int success = 0;

int main(void) {
    int i, j;

    printf("显示迷宫：\n");
    for(i = 0; i < 7; i++) {
        for(j = 0; j < 7; j++)
            if(maze[i][j] == 2)
                printf("■");
            else
                printf("  ");
        printf("\n");
    }

    if(visit(startI, startJ) == 0)
        printf("\n没有找到出口！\n");
    else {
        printf("\n显示路径：\n");
        for(i = 0; i < 7; i++) {
            for(j = 0; j < 7; j++) {
                if(maze[i][j] == 2)
                    printf("■");
                else if(maze[i][j] == 1)
                    printf("◇");
                else
                    printf("  ");
            }
            printf("\n");
        }
    }
```

```
    return 0;
}

int visit(int i, int j) {
    maze[i][j] = 1;

    if(i == endI && j == endJ)
        success = 1;

    if(success != 1 && maze[i][j+1] == 0) visit(i, j+1);
    if(success != 1 && maze[i+1][j] == 0) visit(i+1, j);
    if(success != 1 && maze[i][j-1] == 0) visit(i, j-1);
    if(success != 1 && maze[i-1][j] == 0) visit(i-1, j);

    if(success != 1)
        maze[i][j] = 0;

    return success;
}
```

6.老鼠走迷官（二）

说明由于迷宫的设计，老鼠走迷宫的入口至出口路径可能不只一条，如何求出所有的路径呢？

解法求所有路径看起来复杂但其实更简单，只要在老鼠走至出口时显示经过的路径，然后退回上一格重新选择下一个位置继续递回就可以了，比求出单一路径还简单，我们的程式只要作一点修改就可以了。

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

void visit(int, int);

int maze[9][9] = {{2, 2, 2, 2, 2, 2, 2, 2, 2},
                  {2, 0, 0, 0, 0, 0, 0, 0, 2},
                  {2, 0, 2, 2, 0, 2, 2, 0, 2},
                  {2, 0, 2, 0, 0, 2, 0, 0, 2},
                  {2, 0, 2, 0, 2, 0, 2, 0, 2},
                  {2, 0, 0, 0, 0, 0, 2, 0, 2},
                  {2, 2, 0, 2, 2, 0, 2, 2, 2},
                  {2, 0, 0, 0, 0, 0, 0, 0, 2},
                  {2, 2, 2, 2, 2, 2, 2, 2, 2}};

int startI = 1, startJ = 1;   // 入口
int endI = 7, endJ = 7;   // 出口

int main(void) {
    int i, j;
```

```
    printf("显示迷宫：\n");
    for(i = 0; i < 7; i++) {
        for(j = 0; j < 7; j++)
            if(maze[i][j] == 2)
                printf("■");
            else
                printf("  ");
        printf("\n");
    }

    visit(startI, startJ);

    return 0;
}

void visit(int i, int j) {
    int m, n;

    maze[i][j] = 1;

    if(i == endI && j == endJ) {
        printf("\n显示路径：\n");
        for(m = 0; m < 9; m++) {
            for(n = 0; n < 9; n++)
                if(maze[m][n] == 2)
                    printf("■");
                else if(maze[m][n] == 1)
                    printf("◇");
                else
                    printf("  ");
            printf("\n");
        }
    }

    if(maze[i][j+1] == 0) visit(i, j+1);
    if(maze[i+1][j] == 0) visit(i+1, j);
    if(maze[i][j-1] == 0) visit(i, j-1);
    if(maze[i-1][j] == 0) visit(i-1, j);

    maze[i][j] = 0;
}
```

7.骑士走棋盘

说明骑士旅游（Knight tour）在十八世纪初倍受数学家与拼图迷的注意，它什么时候被提出已不可考，骑士的走法为西洋棋的走法，骑士可以由任一个位置出发，它要如何走完[所有的位置？

解法骑士的走法，基本上可以使用递回来解决，但是纯綷的递回在维度大时相当没有效率，一个聪明的解法由J.C. Warnsdorff

在1823年提出，简单的说，先将最难的位置走完，接下来的路就宽广了，骑士所要走的下一步，「为下一步再选择时，所能走的步数最少的一步。」，使用这个方法，在不使用递回的情况下，可以有较高的机率找出走法（找不到走法的机会也是有的）。

```
#include <stdio.h>

int board[8][8] = {0};

int main(void) {
	int startx, starty;
	int i, j;
	printf("输入起始点：");
	scanf("%d %d", &startx, &starty);

	if(travel(startx, starty)) {
		printf("游历完成！\n");
	}
	else {
		printf("游历失败！\n");
	}

	for(i = 0; i < 8; i++) {
		for(j = 0; j < 8; j++) {
			printf("%2d ", board[i][j]);
		}
		putchar('\n');
	}
	return 0;
}

int travel(int x, int y) {
	// 对应骑士可走的八个方向
	int ktmove1[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
	int ktmove2[8] = {1, 2, 2, 1, -1, -2, -2, -1};

	// 测试下一步的出路
	int nexti[8] = {0};
	int nextj[8] = {0};
	// 记录出路的个数
	int exists[8] = {0};
	int i, j, k, m, l;
	int tmpi, tmpj;
	int count, min, tmp;

	i = x;
	j = y;
	board[i][j] = 1;

	for(m = 2; m <= 64; m++) {
		for(l = 0; l < 8; l++)
			exists[l] = 0;
```

```
l = 0;

// 试探八个方向
for(k = 0; k < 8; k++) {
    tmpi = i + ktmove1[k];
    tmpj = j + ktmove2[k];

    // 如果是边界了，不可走
    if(tmpi < 0 || tmpj < 0 || tmpi > 7 || tmpj > 7)
        continue;

    // 如果这个方向可走，记录下来
    if(board[tmpi][tmpj] == 0) {
        nexti[l] = tmpi;
        nextj[l] = tmpj;
        // 可走的方向加一个
        l++;
    }
}

count = l;
// 如果可走的方向为0个，返回
if(count == 0) {
    return 0;
}
else if(count == 1) {
    // 只有一个可走的方向
    // 所以直接是最少出路的方向
    min = 0;
}
else {
    // 找出下一个位置的出路数
    for(l = 0; l < count; l++) {
        for(k = 0; k < 8; k++) {
            tmpi = nexti[l] + ktmove1[k];
            tmpj = nextj[l] + ktmove2[k];
            if(tmpi < 0 || tmpj < 0 ||
                tmpi > 7 || tmpj > 7) {
                continue;
            }
            if(board[tmpi][tmpj] == 0)
                exists[l]++;
        }
    }
    tmp = exists[0];
    min = 0;
    // 从可走的方向中寻找最少出路的方向
    for(l = 1; l < count; l++) {
        if(exists[l] < tmp) {
            tmp = exists[l];
            min = l;
```

```
                    }
                }
            }

            // 走最少出路的方向
            i = nexti[min];
            j = nextj[min];
            board[i][j] = m;
        }

        return 1;
}
```

8.八皇后

说明西洋棋中的皇后可以直线前进，吃掉遇到的所有棋子，如果棋盘上有八个皇后，则这八个皇后如何相安无事的放置在棋盘上，1970年与1971年， E.W.Dijkstra与N.Wirth曾经用这个问题来讲解程式设计之技巧。

解法关于棋盘的问题，都可以用递回求解，然而如何减少递回的次数？在八个皇后的问题中，不必要所有的格子都检查过，例如若某列检查过，该该列的其它格子就不用再检查了，这个方法称为分支修剪。

```
#include <stdio.h>
#include <stdlib.h>
#define N 8

int column[N+1]; // 同栏是否有皇后，1表示有
int rup[2*N+1]; // 右上至左下是否有皇后
int lup[2*N+1]; // 左上至右下是否有皇后
int queen[N+1] = {0};
int num; // 解答编号

void backtrack(int); // 递回求解

int main(void) {
    int i;
    num = 0;

    for(i = 1; i <= N; i++)
        column[i] = 1;

    for(i = 1; i <= 2*N; i++)
        rup[i] = lup[i] = 1;

    backtrack(1);

    return 0;
}
```

```
void showAnswer() {
    int x, y;
    printf("\n解答  %d\n", ++num);
    for(y = 1; y <= N; y++) {
        for(x = 1; x <= N; x++) {
            if(queen[y] == x) {
                printf(" Q");
            }
            else {
                printf(" .");
            }
        }
        printf("\n");
    }
}

void backtrack(int i) {
    int j;

    if(i > N) {
        showAnswer();
    }
    else {
        for(j = 1; j <= N; j++) {
            if(column[j] == 1 &&
                rup[i+j] == 1 && lup[i-j+N] == 1) {
                queen[i] = j;
                // 设定为占用
                column[j] = rup[i+j] = lup[i-j+N] = 0;
                backtrack(i+1);
                column[j] = rup[i+j] = lup[i-j+N] = 1;
            }
        }
    }
}
```

9.八枚银币

说明现有八枚银币a b c d e f g h，已知其中一枚是假币，其重量不同于真币，但不知是较轻或较重，如何使用天平以最少的比较次数，决定出哪枚是假币，并得知假币比真币较轻或较重。

解法单就求假币的问题是不难，但问题限制使用最少的比较次数，所以我们不能以单纯的回圈比较来求解，我们可以使用决策树（decision tree），使用分析与树状图来协助求解。一个简单的状况是这样的，我们比较a+b+c与d+e+f ，如果相等，则假币必是g或h，我们先比较g或h哪个较重，如果g较重，再与a比较（a是真币），如果g等于a，则g为真币，则h为假币，由于h比g轻而 g是真币，则h假币的重量比真币轻。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void compare(int[], int, int, int);
void eightcoins(int[]);

int main(void) {
	int coins[8] = {0};
	int i;

	srand(time(NULL));

	for(i = 0; i < 8; i++)
		coins[i] = 10;

	printf("\n输入假币重量(比10大或小)：");
	scanf("%d", &i);
	coins[rand() % 8] = i;

	eightcoins(coins);

	printf("\n\n列出所有钱币重量：");
	for(i = 0; i < 8; i++)
		printf("%d ", coins[i]);

	printf("\n");

	return 0;
}

void compare(int coins[], int i, int j, int k) {
	if(coins[i] > coins[k])
		printf("\n假币 %d 较重", i+1);
	else
		printf("\n假币 %d 较轻", j+1);
}

void eightcoins(int coins[]) {
	if(coins[0]+coins[1]+coins[2] ==
	   coins[3]+coins[4]+coins[5]) {
	  if(coins[6] > coins[7])
		compare(coins, 6, 7, 0);
	  else
		compare(coins, 7, 6, 0);
	}
	else if(coins[0]+coins[1]+coins[2] >
		coins[3]+coins[4]+coins[5]) {
	  if(coins[0]+coins[3] == coins[1]+coins[4])
		compare(coins, 2, 5, 0);
	  else if(coins[0]+coins[3] > coins[1]+coins[4])
```

```
            compare(coins, 0, 4, 1);
        if(coins[0]+coins[3] < coins[1]+coins[4])
            compare(coins, 1, 3, 0);
    }
    else if(coins[0]+coins[1]+coins[2] <
            coins[3]+coins[4]+coins[5]) {
        if(coins[0]+coins[3] == coins[1]+coins[4])
            compare(coins, 5, 2, 0);
        else if(coins[0]+coins[3] > coins[1]+coins[4])
            compare(coins, 3, 1, 0);
        if(coins[0]+coins[3] < coins[1]+coins[4])
            compare(coins, 4, 0, 1);
    }
}
```

10.生命游戏

说明生命游戏（game of life）为1970年由英国数学家J. H. Conway所提出，某一细胞的邻居包括上、下、左、右、左上、左下、右上与右下相邻之细胞，游戏规则如下：

孤单死亡：如果细胞的邻居小于一个，则该细胞在下一次状态将死亡。
拥挤死亡：如果细胞的邻居在四个以上，则该细胞在下一次状态将死亡。
稳定：如果细胞的邻居为二个或三个，则下一次状态为稳定存活。
复活：如果某位置原无细胞存活，而该位置的邻居为三个，则该位置将复活一细胞。

解法生命游戏的规则可简化为以下，并使用CASE比对即可使用程式实作：

邻居个数为0、1、4、5、6、7、8时，则该细胞下次状态为死亡。
邻居个数为2时，则该细胞下次状态为复活。
邻居个数为3时，则该细胞下次状态为稳定。

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define MAXROW 10
#define MAXCOL 25
#define DEAD 0
#define ALIVE 1
int map[MAXROW][MAXCOL], newmap[MAXROW][MAXCOL];

void init();
int neighbors(int, int);
void outputMap();
void copyMap();

int main() {
    int row, col;
    char ans;
    init();
    while(1) {
```

```
        outputMap();
        for(row = 0; row < MAXROW; row++) {
            for(col = 0; col < MAXCOL; col++) {
                switch (neighbors(row, col)) {
                    case 0:
                    case 1:
                    case 4:
                    case 5:
                    case 6:
                    case 7:
                    case 8:
                        newmap[row][col] = DEAD;
                        break;
                    case 2:
                        newmap[row][col] = map[row][col];
                        break;
                    case 3:
                        newmap[row][col] = ALIVE;
                        break;
                }
            }
        }

        copyMap();
        printf("\nContinue next Generation ? ");
        getchar();
        ans = toupper(getchar());
        if(ans != 'Y')      break;
    }
    return 0;
}

void init() {
    int row, col;

    for(row = 0; row < MAXROW; row++)
        for(col = 0; col < MAXCOL; col++)
            map[row][col] = DEAD;

    puts("Game of life Program");
    puts("Enter x, y where x, y is living cell");
    printf("0 <= x <= %d, 0 <= y <= %d\n",
                    MAXROW-1, MAXCOL-1);
    puts("Terminate with x, y = -1, -1");

    while(1) {
        scanf("%d %d", &row, &col);
        if(0 <= row && row < MAXROW &&
            0 <= col && col < MAXCOL)
            map[row][col] = ALIVE;
        else if(row == -1 || col == -1)
```

```
            break;
        else
            printf("(x, y) exceeds map ranage!");
    }
}

int neighbors(int row, int col) {
    int count = 0, c, r;
    for(r = row-1; r <= row+1; r++)
        for(c = col-1; c <= col+1; c++) {
            if(r < 0 || r >= MAXROW || c < 0 || c >= MAXCOL)
                continue;
            if(map[r][c] == ALIVE)
                count++;
        }

    if(map[row][col] == ALIVE)
        count--;
    return count;
}

void outputMap() {
    int row, col;
    printf("\n\n%20cGame of life cell status\n");
    for(row = 0; row < MAXROW; row++) {
        printf("\n%20c", ' ');
        for(col = 0; col < MAXCOL; col++)
            if(map[row][col] == ALIVE)      putchar('#');
            else        putchar('-');
    }
}

void copyMap() {
    int row, col;
    for(row = 0; row < MAXROW; row++)
        for(col = 0; col < MAXCOL; col++)
            map[row][col] = newmap[row][col];
}
```

11.字串核对

说明今日的一些高阶程式语言对于字串的处理支援越来越强大（例如Java、Perl等），不过字串搜寻本身仍是个值得探讨的课题，在这边以Boyer- Moore法来说明如何进行字串说明，这个方法快且原理简洁易懂。

解法字串搜寻本身不难，使用暴力法也可以求解，但如何快速搜寻字串就不简单了，传统的字串搜寻是从关键字与字串的开头开始比对，例如 Knuth-Morris-Pratt 演算法 字串搜寻，这个方法也不错，不过要花时间在公式计算上；Boyer-Moore字串核对改由关键字的后面开始核对字串，并制作前进表，如果比对不符合则依前进表中的值前进至下一个核对处，假设是p好了，然后比对字串中p-n+1至p的值是否与关键字相同。

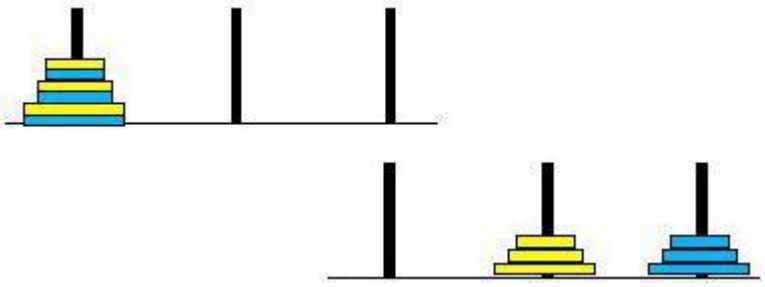
如果关键字中有重复出现的字元，则前进值就会有两个以上的值，此时则取前进值较小的值，如此就不会跳过可能的位置，例如texture这个关键字，t的前进值应该取后面的3而不是取前面的7。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void table(char*); //  建立前进表
int search(int, char*, char*); //  搜寻关键字
void substring(char*, char*, int, int); //  取出子字串

int skip[256];

int main(void) {
	char str_input[80];
	char str_key[80];
	char tmp[80] = {'\0'};
	int m, n, p;
	printf("请输入字串：");
	gets(str_input);
	printf("请输入搜寻关键字：");
	gets(str_key);
	m = strlen(str_input); //  计算字串长度
	n = strlen(str_key);
	table(str_key);
	p = search(n-1, str_input, str_key);

	while(p != -1) {
		substring(str_input, tmp, p, m);
		printf("%s\n", tmp);
		p = search(p+n+1, str_input, str_key);
}

	printf("\n");
	return 0;
}

void table(char *key) {
	int k, n;
	n = strlen(key);
	for(k = 0; k <= 255; k++)
	skip[k] = n;
	for(k = 0; k < n - 1; k++)
	skip[key[k]] = n - k - 1;
}

int search(int p, char* input, char* key) {
	int i, m, n;
	char tmp[80] = {'\0'};
	m = strlen(input);
	n = strlen(key);
```

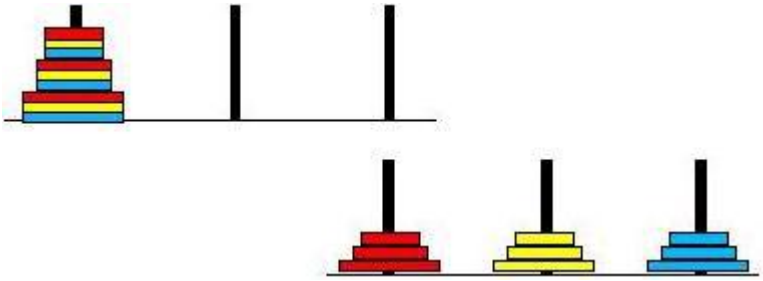
```
    while(p < m) {
        substring(input, tmp, p-n+1, p);
        if(!strcmp(tmp, key)) //  比较两字串是否相同
            return p-n+1;
        p += skip[input[p]];
    }
    return -1;
}

void substring(char *text, char* tmp, int s, int e) {
    int i, j;
    for(i = s, j = 0; i <= e; i++, j++)
    mp[j] = text[i];
    tmp[j] = '\0';
}
```

12.双色、三色河内塔

说明双色河内塔与三色河内塔是由之前所介绍过的河内塔规则衍生而来，双色河内塔的目的是将下图左上的圆环位置经移动成为右下的圆环位置：

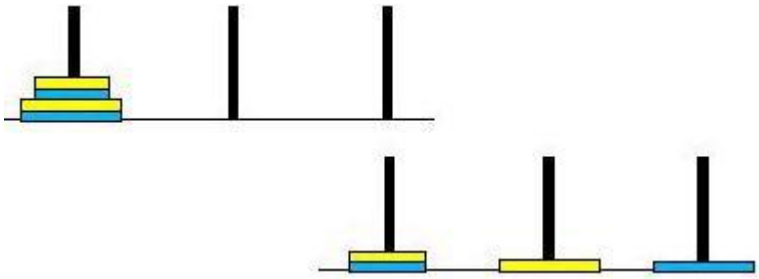


而三色河内塔则是将下图左上的圆环经移动成为右上的圆环：

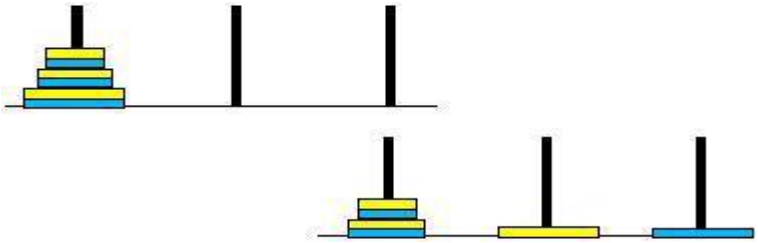


解法无论是双色河内塔或是三色河内塔，其解法观念与之前介绍过的河内塔是类似的，同样也是使用递回来解，不过这次递回解法的目的不同，我们先来看只有两个盘的情况，这很简单，只要将第一柱的黄色移动至第二柱，而接下来第一柱的蓝色移动至第三柱。

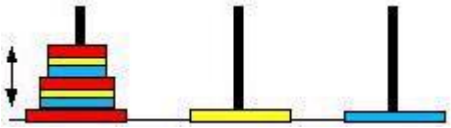
再来是四个盘的情况，首先必须用递回完成下图左上至右下的移动：



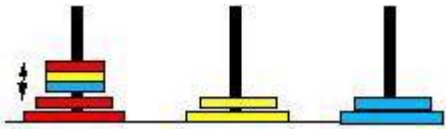
接下来最底层的就不用管它们了，因为它们已经就定位，只要再处理第一柱的上面两个盘子就可以了。那么六个盘的情况呢？一样！首先必须用递回完成下图左上至右下的移动：



接下来最底层的就不用管它们了，因为它们已经就定位，只要再处理第一柱上面的四个盘子就可以了，这又与之前只有四盘的情况相同，接下来您就知道该如何进行解题了，无论是八个盘、十个盘以上等，都是用这个观念来解题。
那么三色河内塔呢？一样，直接来看九个盘的情况，首先必须完成下图的移动结果：



接下来最底两层的就不用管它们了，因为它们已经就定位，只要再处理第一柱上面的三个盘子就可以了。



双色河内塔 C 实作

```
#include <stdio.h>


void hanoi(int disks, char source, char temp, char target) {
	if (disks == 1) {
		printf("move disk from %c to %c\n", source, target);
		printf("move disk from %c to %c\n", source, target);
	} else {
		hanoi(disks-1, source, target, temp);
		hanoi(1, source, temp, target);
		hanoi(disks-1, temp, source, target);
	}
}

void hanoi2colors(int disks) {
	char source = 'A';
	char temp = 'B';
	char target = 'C';
	int i;
	for(i = disks / 2; i > 1; i--) {
		hanoi(i-1, source, temp, target);
		printf("move disk from %c to %c\n", source, temp);
```

```
            printf("move disk from %c to %c\n", source, temp);
            hanoi(i-1, target, temp, source);
            printf("move disk from %c to %c\n", temp, target);
        }
        printf("move disk from %c to %c\n", source, temp);
        printf("move disk from %c to %c\n", source, target);
}

int main() {
        int n;
        printf("请输入盘数： ");
        scanf("%d", &n);

        hanoi2colors(n);

        return 0;
}
```

三色河内塔 C 实作

```
#include <stdio.h>


void hanoi(int disks, char source, char temp, char target) {
        if (disks == 1) {
            printf("move disk from %c to %c\n", source, target);
            printf("move disk from %c to %c\n", source, target);
            printf("move disk from %c to %c\n", source, target);
        } else {
            hanoi(disks-1, source, target, temp);
            hanoi(1, source, temp, target);
            hanoi(disks-1, temp, source, target);
        }
}

void hanoi3colors(int disks) {
        char source = 'A';
        char temp = 'B';
        char target = 'C';
        int i;
        if(disks == 3) {
            printf("move disk from %c to %c\n", source, temp);
            printf("move disk from %c to %c\n", source, temp);
            printf("move disk from %c to %c\n", source, target);
            printf("move disk from %c to %c\n", temp, target);
            printf("move disk from %c to %c\n", temp, source);
            printf("move disk from %c to %c\n", target, temp);;
        }
        else {
            hanoi(disks/3-1, source, temp, target);
            printf("move disk from %c to %c\n", source, temp);
            printf("move disk from %c to %c\n", source, temp);
```

```
        printf("move disk from %c to %c\n", source, temp);

        hanoi(disks/3-1, target, temp, source);
        printf("move disk from %c to %c\n", temp, target);
        printf("move disk from %c to %c\n", temp, target);
        printf("move disk from %c to %c\n", temp, target);

        hanoi(disks/3-1, source, target, temp);
        printf("move disk from %c to %c\n", target, source);
        printf("move disk from %c to %c\n", target, source);

        hanoi(disks/3-1, temp, source, target);
        printf("move disk from %c to %c\n", source, temp);

        for (i = disks / 3 - 1; i > 0; i--) {
            if (i>1) {
                hanoi(i-1, target, source, temp);
            }
            printf("move disk from %c to %c\n",target, source);
            printf("move disk from %c to %c\n",target, source);
            if (i>1) {
                hanoi(i-1, temp, source, target);
            }
            printf("move disk from %c to %c\n", source, temp);
        }
    }
}

int main() {
    int n;
    printf("请输入盘数：");
    scanf("%d", &n);

    hanoi3colors(n);
    return 0;
}
```

13.背包问题（Knapsack Problem）

说明假设有一个背包的负重最多可达8公斤，而希望在背包中装入负重范围内可得之总价物品，假设是水果好了，水果的编号、单价与重量如下所示：

0	李子	4KG	NT$4500
1	苹果	5KG	NT$5700
2	橘子	2KG	NT$2250
3	草莓	1KG	NT$1100
4	甜瓜	6KG	NT$6700

解法背包问题是关于最佳化的问题，要解最佳化问题可以使用「动态规划」（Dynamic programming），从空集合开始，每增加一个元素就先求出该阶段的最佳解，直到所有的元素加入至集合中，最后得到的就是最佳解。

以背包问题为例，我们使用两个阵列value与item，value表示目前的最佳解所得之总价，item表示最后一个放至背包的水果，假设有负重量 1～8的背包8个，并对每个背包求其最佳解。

逐步将水果放入背包中，并求该阶段的最佳解：

放入李子

背包负重	1	2	3	4	5	6	7	8
value	0	0	0	4500	4500	4500	4500	9000
item	—	—	—	0	0	0	0	0

放入苹果

背包负重	1	2	3	4	5	6	7	8
value	0	0	0	4500	5700	5700	5700	9000
item	—	—	—	0	1	1	1	0

放入橘子

背包负重	1	2	3	4	5	6	7	8
value	0	2250	2250	4500	5700	6750	7950	9000
item	—	2	2	0	1	2	2	0

放入草莓

背包负重	1	2	3	4	5	6	7	8
value	1100	2250	3350	4500	5700	6800	7950	9050
item	3	2	3	0	1	3	2	3

放入甜瓜

背包负重	1	2	3	4	5	6	7	8

valu e	110 0	225 0	335 0	450 0	570 0	680 0	795 0	905 0
item	3	2	3	0	1	3	2	3

由最后一个表格，可以得知在背包负重8公斤时，最多可以装入9050元的水果，而最后一个装入的 水果是3号，也就是草莓，装入了草莓，背包只能再放入7公斤（8-1）的水果，所以必须看背包负重7公斤时的最佳解，最后一个放入的是2号，也就 是橘子，现在背包剩下负重量5公斤（7-2），所以看负重5公斤的最佳解，最后放入的是1号，也就是苹果，此时背包负重量剩下0公斤（5-5），无法 再放入水果，所以求出最佳解为放入草莓、橘子与苹果，而总价为9050元。

实作

C

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

#define LIMIT 8     // 重量限制
#define N 5         // 物品种类
#define MIN 1       // 最小重量

struct body {
    char name[20];
    int size;
    int price;
};

typedef struct body object;

int main(void) {
    int item[LIMIT+1] = {0};
    int value[LIMIT+1] = {0};
    int newvalue, i, s, p;

    object a[] = {{"李子", 4, 4500},
                  {"苹果", 5, 5700},
                  {"橘子", 2, 2250},
                  {"草莓", 1, 1100},
                  {"甜瓜", 6, 6700}};

    for(i = 0; i < N; i++) {
        for(s = a[i].size; s <= LIMIT; s++) {
            p = s - a[i].size;
            newvalue = value[p] + a[i].price;
            if(newvalue > value[s]) {// 找到阶段最佳解
                value[s] = newvalue;
                item[s] = i;
            }
        }
    }
```

```
    printf("物品\t价格\n");
    for(i = LIMIT; i >= MIN; i = i - a[item[i]].size) {
        printf("%s\t%d\n",
                a[item[i]].name, a[item[i]].price);
    }

    printf("合计\t%d\n", value[LIMIT]);

    return 0;
}
```

Java

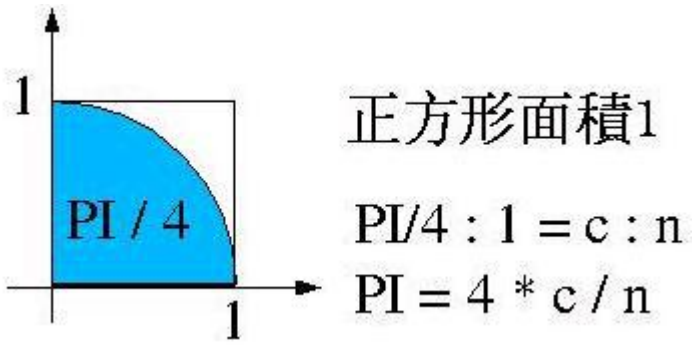
```
class Fruit {
    private String name;
    private int size;
    private int price;

    public Fruit(String name, int size, int price) {
        this.name = name;
        this.size = size;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }

    public int getSize() {
        return size;
    }
}

public class Knapsack {
    public static void main(String[] args) {
        final int MAX = 8;
        final int MIN = 1;
        int[] item = new int[MAX+1];
        int[] value = new int[MAX+1];

        Fruit fruits[] = {
                new Fruit("李子", 4, 4500),
                new Fruit("苹果", 5, 5700),
                new Fruit("橘子", 2, 2250),
                new Fruit("草莓", 1, 1100),
                new Fruit("甜瓜", 6, 6700)};
```

```
        for(int i = 0; i < fruits.length; i++) {
            for(int s = fruits[i].getSize(); s <= MAX; s++) {
                int p = s - fruits[i].getSize();
                int newvalue = value[p] +
                                        fruits[i].getPrice();
                if(newvalue > value[s]) {// 找到阶段最佳解
                    value[s] = newvalue;
                    item[s] = i;
                }
            }
        }

        System.out.println("物品\t价格");
        for(int i = MAX;
            i >= MIN;
            i = i - fruits[item[i]].getSize()) {
            System.out.println(fruits[item[i]].getName()+
                    "\t" + fruits[item[i]].getPrice());
        }

        System.out.println("合计\t" + value[MAX]);
    }
}
```

14.蒙地卡罗法求 PI

说明蒙地卡罗为摩洛哥王国之首都，该国位于法国与义大利国境，以赌博闻名。蒙地卡罗的基本原理为以乱数配合面积公式来进行解题，这种以机率来解题的方式带有赌博的意味，虽然在精确度上有所疑虑，但其解题的思考方向却是个值得学习的方式。

解法蒙地卡罗的解法适用于与面积有关的题目，例如求PI值或椭圆面积，这边介绍如何求PI值；假设有一个圆半径为1，所以四分之一圆面积就为PI，而包括此四分之一圆的正方形面积就为1，如下图所示：



如果随意的在正方形中投射飞标（点）好了，则这些飞标（点）有些会落于四分之一圆内，假设所投射的飞标（点）有n点，在圆内的飞标（点）有c点，则依比例来算，就会得到上图中最后的公式。

至于如何判断所产生的点落于圆内，很简单，令乱数产生X与Y两个数值，如果X^2+Y^2等于1就是落在圆内。

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

```
#include <stdlib.h>
#include <time.h>

#define N 50000

int main(void) {
	int i, sum = 0;
	double x, y;

	srand(time(NULL));

	for(i = 1; i < N; i++) {
		x = (double) rand() / RAND_MAX;
		y = (double) rand() / RAND_MAX;
		if((x * x + y * y) < 1)
			sum++;
	}
	printf("PI = %f\n", (double) 4 * sum / N);
	return 0;
}
```

15.Eratosthenes 筛选求质数

说明除了自身之外，无法被其它整数整除的数称之为质数，要求质数很简单，但如何快速的求出质数则一直是程式设计人员与数学家努力的课题，在这边介绍一个着名的 Eratosthenes求质数方法。

解法首先知道这个问题可以使用回圈来求解，将一个指定的数除以所有小于它的数，若可以整除就不是质数，然而如何减少回圈的检查次数？如何求出小于N的所有质数？

首先假设要检查的数是N好了，则事实上只要检查至N的开根号就可以了，道理很简单，假设A*B = N，如果A大于N的开根号，则事实上在小于A之前的检查就可以先检查到B这个数可以整除N。不过在程式中使用开根号会精确度的问题，所以可以使用i*i <= N进行检查，且执行更快。

再来假设有一个筛子存放1～N，例如：

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 ..... N

先将2的倍数筛去：

2 3 5 7 9 11 13 15 17 19 21 ..... N

再将3的倍数筛去：

2 3 5 7 11 13 17 19 ..... N

再来将5的倍数筛去，再来将7的质数筛去，再来将11的倍数筛去.....，如此进行到最后留下的数就都是质数，这就是Eratosthenes筛选方法（Eratosthenes Sieve Method）。

检查的次数还可以再减少，事实上，只要检查6n+1与6n+5就可以了，也就是直接跳过2与3的倍数，使得程式中的if的检查动作可以减少。

实作

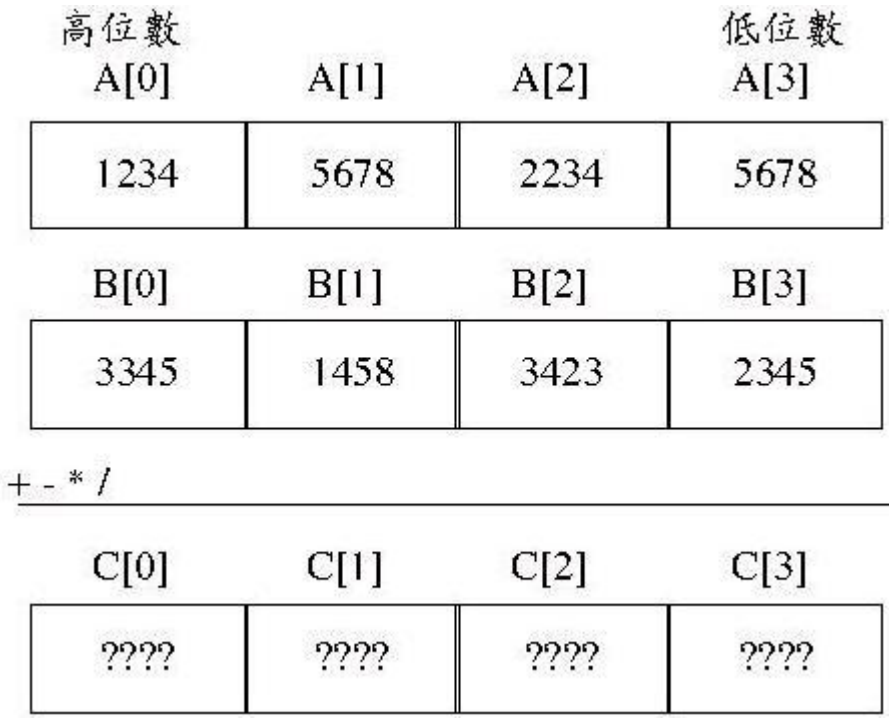
C

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

#define N 1000

int main(void) {
    int i, j;
    int prime[N+1];

    for(i = 2; i <= N; i++)
        prime[i] = 1;

    for(i = 2; i*i <= N; i++) { //  这边可以改进
        if(prime[i] == 1) {
            for(j = 2*i; j <= N; j++) {
                if(j % i == 0)
                    prime[j] = 0;
            }
        }
    }

    for(i = 2; i < N; i++) {
        if(prime[i] == 1) {
            printf("%4d ", i);
            if(i % 16 == 0)
                printf("\n");
        }
    }

    printf("\n");
    return 0;
}
```

16.超长整数运算（大数运算）

说明基于记忆体的有效运用，程式语言中规定了各种不同的资料型态，也因此变数所可以表达的最大整数受到限制，例如12345678912345678 9这样的 整数就不可能储存在long变数中（例如C/C++等），我们称这为long数，这边翻为超长整数（避免与资料型态的长整数翻译混淆），或俗称大数运算。

解法一个变数无法表示超长整数，则就使用多个变数，当然这使用阵列最为方便，假设程式语言的最大资料型态可以储存至65535的数好了，为了计算方便及符合使用十进位制的习惯，让每一个阵列元素可以储存四个位数，也就是0到9999的数，例如：



很多人问到如何计算像50!这样的问题，解法就是使用程式中的乘法函式，至于要算到多大，就看需求了。

由于使用阵列来储存数值，关于数值在运算时的加减乘除等各种运算、位数的进位或借位就必须自行定义，加、减、乘都是由低位数开始运算，而除法则是由高位数开始运算，这边直接提供加减乘除运算的函式供作参考，以下的N为阵列长度。

```
void add(int *a, int *b, int *c) {
    int i, carry = 0;

    for(i = N - 1; i >= 0; i--) {
        c[i] = a[i] + b[i] + carry;
        if(c[i] < 10000)
            carry = 0;
        else { // 进位
            c[i] = c[i] - 10000;
            carry = 1;
        }
    }
}

void sub(int *a, int *b, int *c) {
    int i, borrow = 0;
    for(i = N - 1; i >= 0; i--) {
        c[i] = a[i] - b[i] - borrow;
        if(c[i] >= 0)
            borrow = 0;
        else { // 借位
            c[i] = c[i] + 10000;
            borrow = 1;
        }
    }
}
```

```
void mul(int *a, int b, int *c) { // b 为乘数
    int i, tmp, carry = 0;
    for(i = N - 1; i >=0; i--) {
        tmp = a[i] * b + carry;
        c[i] = tmp % 10000;
        carry = tmp / 10000;
    }
}

void div(int *a, int b, int *c) {   // b 为除数
    int i, tmp, remain = 0;
    for(i = 0; i < N; i++) {
        tmp = a[i] + remain;
        c[i] = tmp / b;
        remain = (tmp % b) * 10000;
    }
}
```

17.长 PI

说明圆周率后的小数位数是无止境的，如何使用电脑来计算这无止境的小数是一些数学家与程式设计师所感兴趣的，在这边介绍一个公式配合 大数运算，可以计算指定位数的圆周率。

解法首先介绍J.Marchin的圆周率公式：

$$PI = [16/5 - 16 / (3*5^3) + 16 / (5*5^5) - 16 / (7*5^7) + ...] -$$

$$[4/239 - 4/(3*239^3) + 4/(5*239^5) - 4/(7*239^7) + ...]$$

可以将这个公式整理为：

$$PI = [16/5 - 4/239] - [16/(5^3) - 4/(239^3)]/3 + [16/(5^5) - 4/(239^5)]/5 + ...$$

也就是说第n项，若为奇数则为正数，为偶数则为负数，而项数表示方式为：

$$[16/5^{2*n-1} - 4/239^{2*n-1}] / (2*n-1)$$

如果我们要计算圆周率至10的负L次方，由于$[16/5^{2*n-1} - 4/239^{2*n-1}]$中$16/5^{2*n-1}$比$4/239^{2*n-1}$来的大，具有决定性，所以表示至少必须计算至第n项：

$$[16/5^{2*n-1}] / (2*n-1) = 10^{-L}$$

将上面的等式取log并经过化简，我们可以求得：

$$n = L / (2\log 5) = L / 1.39794$$

所以若要求精确度至小数后L位数，则只要求至公式的第n项，其中n等于：

$$n = [L/1.39794] + 1$$

在上式中[]为高斯符号，也就是取至整数（不大于L/1.39794的整数）；为了计简方便，可以在程式中使用下面这个公式来计简第n项：

$[W_n-1/5^2- V_n-1 / (239^2)] / (2*n-1)$

这个公式的演算法配合大数运算函式的演算法为： div(w, 25, w);
div(v, 239, v);
div(v, 239, v);
sub(w, v, q);
div(q, 2*k-1, q)

至于大数运算的演算法，请参考之前的文章，必须注意的是在输出时，由于是输出阵列中的整数值，如果阵列中整数位数不满四位，则必须补上0，在C语言中只要 使用格式指定字%04d，使得不足位数部份自动补上0再输出，至于Java的部份，使用 NumberFormat来作格式化。

```
#include <stdio.h>

#define L 1000
#define N L/4+1

// L 为位数，N是array长度

void add(int*, int*, int*);
void sub(int*, int*, int*);
void div(int*, int, int*);

int main(void) {
    int s[N+3] = {0};
    int w[N+3] = {0};
    int v[N+3] = {0};
    int q[N+3] = {0};
    int n = (int)(L/1.39793 + 1);
    int k;

    w[0] = 16*5;
    v[0] = 4*239;

    for(k = 1; k <= n; k++) {
        // 套用公式
        div(w, 25, w);
        div(v, 239, v);
        div(v, 239, v);
        sub(w, v, q);
        div(q, 2*k-1, q);

        if(k%2) // 奇数项
            add(s, q, s);
        else      // 偶数项
            sub(s, q, s);
    }
```

```
    printf("%d.", s[0]);
    for(k = 1; k < N; k++)
        printf("%04d", s[k]);
    printf("\n");
    return 0;
}

void add(int *a, int *b, int *c) {
    int i, carry = 0;

    for(i = N+1; i >= 0; i--) {
        c[i] = a[i] + b[i] + carry;
        if(c[i] < 10000)
            carry = 0;
        else { // 进位
            c[i] = c[i] - 10000;
            carry = 1;
        }
    }
}

void sub(int *a, int *b, int *c) {
    int i, borrow = 0;
    for(i = N+1; i >= 0; i--) {
        c[i] = a[i] - b[i] - borrow;
        if(c[i] >= 0)
            borrow = 0;
        else { // 借位
            c[i] = c[i] + 10000;
            borrow = 1;
        }
    }
}

void div(int *a, int b, int *c) {   // b 为除数
    int i, tmp, remain = 0;
    for(i = 0; i <= N+1; i++) {
        tmp = a[i] + remain;
        c[i] = tmp / b;
        remain = (tmp % b) * 10000;
    }
}
```

18.最大公因数、最小公倍数、因式分解

说明最大公因数使用辗转相除法来求，最小公倍数则由这个公式来求：

GCD * LCM = 两数乘积

解法最大公因数可以使用递回与非递回求解，因式分解基本上就是使用小于输入数的数值当作除数，去除以输入数值，如

果可以整除就视为因数，要比较快的解法就是求出小于该数的所有质数，并试试看是不是可以整除，求质数的问题是另一个课题，请参考 Eratosthenes 筛选求质数。

实作（最大公因数、最小公倍数）

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

int main(void) {
	int m, n, r;
	int s;

	printf("输入两数：");
	scanf("%d %d", &m, &n);
	s = m * n;

	while(n != 0) {
		r = m % n;
		m = n;
		n = r;
	}

	printf("GCD：%d\n", m);
	printf("LCM：%d\n", s/m);

	return 0;
}
```

实作（因式分解）

C（不用质数表）

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

int main(void) {
	int i, n;

	printf("请输入整数：");
	scanf("%d", &n);
	printf("%d = ", n);

	for(i = 2; i * i <= n;) {
		if(n % i == 0) {
			printf("%d * ", i);
			n /= i;
		}
		else
			i++;
	}
```

```
    printf("%d\n", n);

    return 0;
}
```

C（使用质数表）

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

#define N 1000

int prime(int*);   // 求质数表
void factor(int*, int);   // 求factor

int main(void) {
    int ptable[N+1] = {0};
    int count, i, temp;

    count = prime(ptable);

    printf("请输入一数：");
    scanf("%d", &temp);
    factor(ptable, temp);
    printf("\n");
    return 0;
 }

 int prime(int* pNum) {
    int i, j;
    int prime[N+1];
    for(i = 2; i <= N; i++)
        prime[i] = 1;

    for(i = 2; i*i <= N; i++) {
        if(prime[i] == 1) {
            for(j = 2*i; j <= N; j++) {
                if(j % i == 0)
                    prime[j] = 0;
            }
        }
    }

    for(i = 2, j = 0; i < N; i++) {
        if(prime[i] == 1)
            pNum[j++] = i;
    }
    return j;
}
```

```
void factor(int* table, int num) {
    int i;

    for(i = 0; table[i] * table[i] <= num;) {
        if(num % table[i] == 0) {
            printf("%d * ", table[i]);
            num /= table[i];
        }
        else
            i++;
    }
    printf("%d\n", num);
}
```

19.完美数

说明如果有一数n，其真因数（Proper factor）的总和等于n，则称之为完美数（Perfect Number），例如以下几个数都是完美数：

6 = 1 + 2 + 3

28 = 1 + 2 + 4 + 7 + 14

496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248

程式基本上不难，第一眼看到时会想到使用回圈求出所有真因数，再进一步求因数和，不过若n值很大，则此法会花费许多时间在回圈测试上，十分没有效率，例如求小于10000的所有完美数。

解法如何求小于10000的所有完美数？并将程式写的有效率？基本上有三个步骤：

求出一定数目的质数表
利用质数表求指定数的因式分解
利用因式分解求所有真因数和，并检查是否为完美数

步骤一 与 步骤二 在之前讨论过了，问题在步骤三，如何求真因数和？方法很简单，要先知道将所有真因数和加上该数本身，会等于该数的两倍，例如：

2 * 28 = 1 + 2 + 4 + 7 + 14 + 28

等式后面可以化为：

$2 * 28 = (2^0 + 2^1 + 2^2) * (7^0 + 7^1)$

所以只要求出因式分解，就可以利用回圈求得等式后面的值，将该值除以2就是真因数和了；等式后面第一眼看时可能想到使用等比级数公式来解，不过会使用到次方运算，可以在回圈走访因式分解阵列时，同时计算出等式后面的值，这在下面的实作中可以看到。

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

#define N 1000
#define P 10000
```

```
int prime(int*);    // 求质数表
int factor(int*, int, int*);    // 求factor
int fsum(int*, int);    // sum ot proper factor

int main(void) {
        int ptable[N+1] = {0}; // 储存质数表
        int fact[N+1] = {0};     // 储存因式分解结果
        int count1, count2, i;

        count1 = prime(ptable);

        for(i = 0; i <= P; i++) {
                count2 = factor(ptable, i, fact);
                if(i == fsum(fact, count2))
                        printf("Perfect Number: %d\n", i);
        }

        printf("\n");

        return 0;
}

int prime(int* pNum) {
        int i, j;
        int prime[N+1];

        for(i = 2; i <= N; i++)
                prime[i] = 1;

        for(i = 2; i*i <= N; i++) {
                if(prime[i] == 1) {
                        for(j = 2*i; j <= N; j++) {
                                if(j % i == 0)
                                        prime[j] = 0;
                        }
                }
        }

        for(i = 2, j = 0; i < N; i++) {
                if(prime[i] == 1)
                        pNum[j++] = i;
        }

        return j;
}

int factor(int* table, int num, int* frecord) {
        int i, k;

        for(i = 0, k = 0; table[i] * table[i] <= num;) {
                if(num % table[i] == 0) {
```

```
                frecord[k] = table[i];
                k++;
                num /= table[i];
            }
            else
                i++;
        }

        frecord[k] = num;

        return k+1;
}

int fsum(int* farr, int c) {
        int i, r, s, q;

        i = 0;
        r = 1;
        s = 1;
        q = 1;

        while(i < c) {
            do {
                r *= farr[i];
                q += r;
                i++;
            } while(i < c-1 && farr[i-1] == farr[i]);
            s *= q;
            r = 1;
            q = 1;
        }

        return s / 2;
}
```

20.阿姆斯壮数

说明

在三位的整数中，例如153可以满足$1^3 + 5^3 + 3^3 = 153$，这样的数称之为Armstrong数，试写出一程式找出所有的三位数Armstrong数。

解法

Armstrong数的寻找，其实就是在问如何将一个数字分解为个位数、十位数、百位数.....，这只要使用除法与余数运算就可以了，例如输入 input为abc，则：

```
a = input / 100
b = (input%100) / 10
c = input % 10
```

```
#include <stdio.h>
#include <time.h>
#include <math.h>

int main(void) {
	int a, b, c;
	int input;

	printf("寻找Armstrong数：\n");

	for(input = 100; input <= 999; input++) {
		a = input / 100;
		b = (input % 100) / 10;
		c = input % 10;
		if(a*a*a + b*b*b + c*c*c == input)
			printf("%d ", input);
	}

	printf("\n");

	return 0;
}
```

21.最大访客数

说明

现将举行一个餐会，让访客事先填写到达时间与离开时间，为了掌握座位的数目，必须先估计不同时间的最大访客数。

解法

这个题目看似有些复杂，其实相当简单，单就计算访客数这个目的，同时考虑同一访客的来访时间与离开时间，反而会使程式变得复杂；只要将来访时间与离开时间分开处理就可以了，假设访客 i 的来访时间为x[i]，而离开时间为y[i]。

在资料输入完毕之后，将x[i]与y[i]分别进行排序（由小到大），道理很简单，只要先计算某时之前总共来访了多少访客，然后再减去某时之前的离开访客，就可以轻易的解出这个问题。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int partition(int[], int, int);
void quicksort(int[], int, int); //  快速排序法
int maxguest(int[], int[], int, int);
```

```
int main(void) {
	int x[MAX] = {0};
	int y[MAX] = {0};
	int time = 0;
	int count = 0;

	printf("\n输入来访与离开125;时间(0~24)：");
	printf("\n范例：10 15");
	printf("\n输入-1 -1结束");
	while(count < MAX) {
		printf("\n>>");
		scanf("%d %d", &x[count], &y[count]);
		if(x[count] < 0)
			break;
		count++;
	}

	if(count >= MAX) {
		printf("\n超出最大访客数(%d)", MAX);
		count--;
	}

	// 预先排序
	quicksort(x, 0, count);
	quicksort(y, 0, count);

	while(time < 25) {
		printf("\n%d 时的最大访客数：%d",
				time, maxguest(x, y, count, time));
		time++;
	}

	printf("\n");

	return 0;
}

int maxguest(int x[], int y[], int count, int time) {
	int i, num = 0;

	for(i = 0; i <= count; i++) {
		if(time > x[i])
			num++;
		if(time > y[i])
			num--;
	}

	return num;
}

int partition(int number[], int left, int right) {
```

```
    int i, j, s;

    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
    }

    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

22.中序式转后序式（前序式）

说明平常所使用的运算式，主要是将运算元放在运算子的两旁，例如a+b/d这样的式子，这称之为中序（Infix）表示式，对于人类来说，这样的式子很容易理 解，但由于电脑执行指令时是有顺序的，遇到中序表示式时，无法直接进行运算，而必须进一步判断运算的先后顺序，所以必须将中序表示式转换为另一种表示方 法。

可以将中序表示式转换为后序（Postfix）表示式，后序表示式又称之为逆向波兰表示式（Reverse polish notation），它是由波兰的数学家卢卡谢维奇提出，例如(a+b)*(c+d)这个式子，表示为后序表示式时是ab+cd+*。

解法用手算的方式来计算后序式相当的简单，将运算子两旁的运算元依先后顺序全括号起来，然后将所有的右括号取代为左边最接近的运算子（从最内层括号开始），最后去掉所有的左括号就可以完成后序表示式，例如：

```
a+b*d+c/d  =>  ((a+(b*d))+(c/d)) -> bd*+cd/+
```

如果要用程式来进行中序转后序，则必须使用堆叠，演算法很简单，直接叙述的话就是使用回圈，取出中序式的字元，遇运算元直接输出，堆叠运算子与左括号， ISP>ICP的话直接输出堆叠中的运算子，遇右括号输出堆叠中的运算子至左括号。

例 如 (a+b)*(c+d) 这个式子，依演算法的输出过程如下： OP	STACK	OUTPUT
(	(	-
a	(	a

+	(+	a
b	(+	ab
)	-	ab+
*	*	ab+
(	*(	ab+
c	*(	ab+c
+	*(+	ab+c
d	*(+	ab+cd
)	*	ab+cd+
-	-	ab+cd+*

如果要将中序式转为前序式，则在读取中序式时是由后往前读取，而左右括号的处理方式相反，其余不变，但输出之前必须先置入堆叠，待转换完成后再将堆叠中的 值由上往下读出，如此就是前序表示式。

实作

C

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

int postfix(char*); // 中序转后序
int priority(char); // 决定运算子优先顺序

int main(void) {
    char input[80];

    printf("输入中序运算式：");
    scanf("%s", input);
    postfix(input);

    return 0;
}

int postfix(char* infix) {
    int i = 0, top = 0;
    char stack[80] = {'\0'};
    char op;

    while(1) {
        op = infix[i];

        switch(op) {
            case '\0':
                while(top > 0) {
                    printf("%c", stack[top]);
                    top--;
                }
                printf("\n");
                return 0;
```

```
            // 运算子堆叠
            case '(':
                if(top < (sizeof(stack) / sizeof(char))) {
                    top++;
                    stack[top] = op;
                }
                break;
            case '+': case '-': case '*': case '/':
                while(priority(stack[top]) >= priority(op)) {
                    printf("%c", stack[top]);
                    top--;
                }
                // 存入堆叠
                if(top < (sizeof(stack) / sizeof(char))) {
                    top++;
                    stack[top] = op;
                }
                break;
            // 遇 ) 输出至 (
            case ')':
                while(stack[top] != '(') {
                    printf("%c", stack[top]);
                    top--;
                }
                top--;   // 不输出(
                break;
            // 运算元直接输出
            default:
                printf("%c", op);
                break;
        }
        i++;
    }
}

int priority(char op) {
    int p;

    switch(op) {
        case '+': case '-':
            p = 1;
            break;
        case '*': case '/':
            p = 2;
            break;
        default:
            p = 0;
            break;
    }

    return p;
```

}

23.后序式的运算

说明 将中序式转换为后序式的好处是，不用处理运算子先后顺序问题，只要依序由运算式由前往后读取即可。

解法

运算时由后序式的前方开始读取，遇到运算元先存入堆叠，如果遇到运算子，则由堆叠中取出两个运算元进行对应的运算，然后将结果存回堆叠，如果运算式读取完 毕，那么堆叠顶的值就是答 案 了 ， 例 如 我 们 计 算 12+34+*这个运算式（也就是(1+2)*(3+4)）： 读取	堆叠
1	1
2	1 2
+	3 // 1+2 后存回
3	3 3
4	3 3 4
+	3 7 // 3+4 后存回
*	21 // 3 * 7 后存回

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

void evalPf(char*);
double cal(double, char, double);

int main(void) {
	char input[80];
	printf("输入后序式：");
	scanf("%s", input);
	evalPf(input);
	return 0;
}
```

```
void evalPf(char* postfix) {
	double stack[80] = {0.0};
	char temp[2];
	char token;
	int top = 0, i = 0;

	temp[1] = '\0';

	while(1) {
		token = postfix[i];
		switch(token) {
			case '\0':
				printf("ans = %f\n", stack[top]);
				return;
			case '+': case '-': case '*': case '/':
				stack[top-1] =
					cal(stack[top], token, stack[top-1]);
				top--;
				break;
			default:
				if(top < sizeof(stack) / sizeof(float)) {
					temp[0] = postfix[i];
					top++;
					stack[top] = atof(temp);
				}
				break;
		}
		i++;
	}
}

double cal(double p1, char op, double p2) {
	switch(op) {
		case '+':
			return p1 + p2;
		case '-':
			return p1 - p2;
		case '*':
			return p1 * p2;
		case '/':
			return p1 / p2;
	}
}
```

24.洗扑克牌（乱数排列）

说明

洗扑克牌的原理其实与乱数排列是相同的，都是将一组数字（例如1～N）打乱重新排列，只不过洗扑克牌多了一个花色判断

的动作而已。

解法

初学者通常会直接想到，随机产生1～N的乱数并将之存入阵列中，后来产生的乱数存入阵列前必须先检查阵列中是否已有重复的数字，如果有这个数就不存入，再重新产生下一个数，运气不好的话，重复的次数就会很多，程式的执行速度就很慢了，这不是一个好方法。

以1～52的乱数排列为例好了，可以将阵列先依序由1到52填入，然后使用一个回圈走访阵列，并随机产生1～52的乱数，将产生的乱数当作索引取出阵列值，并与目前阵列走访到的值相交换，如此就不用担心乱数重复的问题了，阵列走访完毕后，所有的数字也就重新排列了。

至于如何判断花色？这只是除法的问题而已，取商数判断花色，取余数判断数字，您可以直接看程式比较清楚。

实作

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 52

int main(void) {
    int poker[N + 1];
    int i, j, tmp, remain;

    // 初始化阵列
    for(i = 1; i <= N; i++)
        poker[i] = i;

    srand(time(0));

    // 洗牌
    for(i = 1; i <= N; i++) {
        j = rand() % 52 + 1;
        tmp = poker[i];
        poker[i] = poker[j];
        poker[j] = tmp;
    }

    for(i = 1; i <= N; i++) {
        // 判断花色
        switch((poker[i]-1) / 13) {
            case 0:
                printf("桃"); break;
            case 1:
                printf("心"); break;
            case 2:
                printf("砖"); break;
            case 3:
                printf("梅"); break;
```

```
            }

            // 扑克牌数字
            remain = poker[i] % 13;
            switch(remain) {
                case 0:
                    printf("K "); break;
                case 12:
                    printf("Q "); break;
                case 11:
                    printf("J "); break;
                default:
                    printf("%d ", remain); break;
            }

            if(i % 13 == 0)
                printf("\n");
        }

        return 0;
}
```

25.Craps 赌博游戏

说明一个简单的赌博游戏，游戏规则如下：玩家掷两个骰子，点数为1到6，如果第一次点数和为7或11，则玩家胜，如果点数和为2、3或12，则玩家输，如果和 为其它点数，则记录第一次的点数和，然后继续掷骰，直至点数和等于第一次掷出的点数和，则玩家胜，如果在这之前掷出了点数和为7，则玩家输。

解法 规则看来有些复杂，但是其实只要使用switch配合if条件判断来撰写即可，小心不要弄错胜负顺序即可。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define WON 0
#define LOST 1
#define CONTINUE 2

int rollDice() {
    return (rand() % 6) + (rand() % 6) + 2;
}

int main(void) {
    int firstRoll = 1;
    int gameStatus = CONTINUE;
    int die1, die2, sumOfDice;
    int firstPoint = 0;
    char c;
```

```
srand(time(0));

printf("Craps赌博游戏，按Enter键开始游戏****");

while(1) {
      getchar();

    if(firstRoll) {
        sumOfDice = rollDice();
        printf("\n玩家掷出点数和：%d\n", sumOfDice);

        switch(sumOfDice) {
            case 7: case 11:
                gameStatus = WON; break;
            case 2: case 3: case 12:
                gameStatus = LOST; break;
            default:
                firstRoll = 0;
                gameStatus = CONTINUE;
                firstPoint = sumOfDice;
                break;
        }
    }
    else {
        sumOfDice = rollDice();
        printf("\n玩家掷出点数和：%d\n", sumOfDice);

        if(sumOfDice == firstPoint)
            gameStatus = WON;
        else if(sumOfDice == 7)
            gameStatus = LOST;
    }

    if(gameStatus == CONTINUE)
        puts("未分胜负，再掷一次****\n");
    else {
        if(gameStatus == WON)
            puts("玩家胜");
        else
            puts("玩家输");

        printf("再玩一次？ ");
        scanf("%c", &c);
        if(c == 'n') {
            puts("游戏结束");
            break;
        }
        firstRoll = 1;
    }
}
```

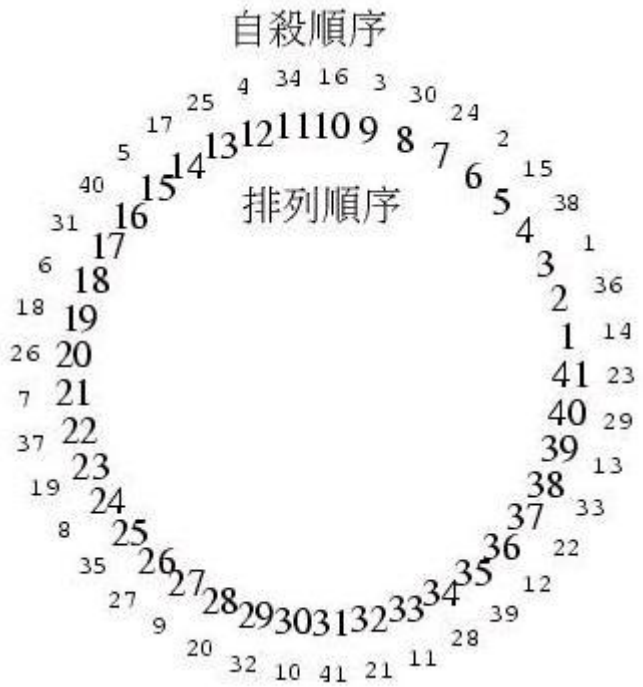
```
    return 0;
}
```

26.约瑟夫问题（Josephus Problem）

说明据说着名犹太历史学家 Josephus有过以下的故事：在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人 开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。

然而Josephus 和他的朋友并不想遵从，Josephus要他的朋友先假装遵从，他将朋友与自己安排在第16个与第31个位置，于是逃过了这场死亡游戏。

解法约瑟夫问题可用代数分析来求解，将这个问题扩大好了，假设现在您与m个朋友不幸参与了这个游戏，您要如何保护您与您的朋友？只要画两个圆圈就可以让自己与朋友免于死亡游戏，这两个圆圈内圈是排列顺序，而外圈是自杀顺序，如下图所示：



使用程式来求解的话，只要将阵列当作环状来处理就可以了，在阵列中由计数1开始，每找到三个无资料区就填入一个计数，直而计数达41为止，然后将阵列由索引1开始列出，就可以得知每个位置的自杀顺序，这就是约瑟夫排列，41个人而报数3的约琴夫排列如下所示：

14 36 1 38 15 2 24 30 3 16 34 4 25 17 5 40 31 6 18 26 7 37 19 8 35 27 9 20 32 10 41 21 11 28 39 12 22 33 13 29 23

由上可知，最后一个自杀的是在第31个位置，而倒数第二个自杀的要排在第16个位置，之前的人都死光了，所以他们也就不知道约琴夫与他的朋友并没有遵守游戏规则了。

```
#include <stdio.h>
#include <stdlib.h>
#define N 41
#define M 3

int main(void) {
```

```
int man[N] = {0};
int count = 1;
int i = 0, pos = -1;
int alive = 0;

while(count <= N) {
    do {
        pos = (pos+1) % N;   // 环状处理
        if(man[pos] == 0)
            i++;

        if(i == M) {   // 报数为3了
            i = 0;
            break;
        }
    } while(1);

    man[pos] = count;
    count++;
}

printf("\n约琴夫排列：");
for(i = 0; i < N; i++)
    printf("%d ", man[i]);
printf("\n\n您想要救多少人？");
scanf("%d", &alive);

printf("\nL表示这%d人要放的位置：\n", alive);
for(i = 0; i < N; i++) {
    if(man[i] > alive)    printf("D");
    else          printf("L");
    if((i+1) % 5 == 0)   printf("    ");
}
printf("\n");
return 0; }
```

27.排列组合

说明将一组数字、字母或符号进行排列，以得到不同的组合顺序，例如1 2 3这三个数的排列组合有：1 2 3、1 3 2、2 1 3、2 3 1、3 1 2、3 2 1。

解法可以使用递回将问题切割为较小的单元进行排列组合，例如1 2 3 4的排列可以分为1 [2 3 4]、2 [1 3 4]、3 [1 2 4]、4 [1 2 3]进行排列，这边利用旋转法，先将旋转间隔设为0，将最右边的数字旋转至最左边，并逐步增加旋转的间隔，例如：

1 2 3 4 -> 旋转1 -> 继续将右边2 3 4进行递回处理
2 1 3 4 -> 旋转1 2 变为 2 1-> 继续将右边1 3 4进行递回处理
3 1 2 4 -> 旋转1 2 3变为 3 1 2 -> 继续将右边1 2 4进行递回处理
4 1 2 3 -> 旋转1 2 3 4变为4 1 2 3 -> 继续将右边1 2 3进行递回处理

```
#include <stdio.h>
#include <stdlib.h>
#define N 4

void perm(int*, int);

int main(void) {
	int num[N+1], i;
	for(i = 1; i <= N; i++)
		num[i] = i;
	perm(num, 1);
	return 0;
}

void perm(int* num, int i) {
	int j, k, tmp;

	if(i < N) {
		for(j = i; j <= N; j++) {
			tmp = num[j];
			// 旋转该区段最右边数字至最左边
			for(k = j; k > i; k--)
				num[k] = num[k-1];
			num[i] = tmp;
			perm(num, i+1);
			// 还原
			for(k = i; k < j; k++)
				num[k] = num[k+1];
			num[j] = tmp;
		}
	}
	else {	// 显示此次排列
		for(j = 1; j <= N; j++)
			printf("%d ", num[j]);
		printf("\n");
	}
}
```

28.格雷码（Gray Code）

说明

Gray Code是一个数列集合，每个数使用二进位来表示，假设使用n位元来表示每个数好了，任两个数之间只有一个位元值不同，例如以下为3位元的Gray Code：

000 001 011 010 110 111 101 100

由定义可以知道，Gray Code的顺序并不是唯一的，例如将上面的数列反过来写，也是一组Gray Code：
100 101 111 110 010 011 001 000

Gray Code是由贝尔实验室的Frank Gray在1940年代提出的，用来在使用PCM（Pusle Code Modulation）方法传送讯号时避免出错，并于1953年三月十七日取得美国专利。

解法

由于Gray Code相邻两数之间只改变一个位元，所以可观 察Gray Code从1变0或从0变1时的位置，假设有4位元的Gray Code如下：
0000 0001 0011 0010 0110 0111 0101 0100
1100 1101 1111 1110 1010 1011 1001 1000

观察奇数项的变化时，我们发现无论它是第几个Gray Code，永远只改变最右边的位元，如果是1就改为0，如果是0就改为1。

观察偶数项的变化时，我们发现所改变的位元，是由右边算来第一个1的左边位元。

以上两个变化规则是固定的，无论位元数为何；所以只要判断位元的位置是奇数还是偶数，就可以决定要改变哪一个位元的值，为了程式撰写方便，将阵列索引 0当作最右边的值，而在列印结果时，是由索引数字大的开始反向列印。

将2位元的Gray Code当作平面座标来看，可以构成一个四边形，您可以发现从任一顶点出发，绕四边形周长绕一圈，所经过的顶点座标就是一组Gray Code，所以您可以得到四组Gray Code。

同样的将3位元的Gray Code当作平面座标来看的话，可以构成一个正立方体，如果您可以从任一顶点出发，将所有的边长走过，并不重复经过顶点的话，所经过的顶点座标顺序之组合也就是一组Gray Code。

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

#define MAXBIT 20
#define TRUE 1
#define CHANGE_BIT(x) x = ((x) == '0' ? '1' : '0')
#define NEXT(x) x = (1 - (x))

int main(void) {
    char digit[MAXBIT];
    int i, bits, odd;

    printf("输入位元数：");
    scanf("%d", &bits);

    for(i = 0; i < bits; i++) {
        digit[i] = '0';
        printf("0");
    }

    printf("\n");

    odd = TRUE;

    while(1) {
```

```
        if(odd)
            CHANGE_BIT(digit[0]);
        else {
            // 计算第一个1的位置
            for(i = 0; i < bits && digit[i] == '0'; i++) ;
            if(i == bits - 1) // 最后一个Gray Code
                break;
            CHANGE_BIT(digit[i+1]);
        }
        for(i = bits - 1; i >= 0; i--)
            printf("%c", digit[i]);

        printf("\n");
        NEXT(odd);
    }
    return 0;
}
```

29.产生可能的集合

说明

给定一组数字或符号，产生所有可能的集合（包括空集合），例如给定1 2 3，则可能的集合为：{}、{1}、{1,2}、{1,2,3}、{1,3}、{2}、{2,3}、{3}。

解法

如果不考虑字典顺序，则有个简单的方法可以产生所有的集合，思考二进位数字加法，并注意1出现的位置，如果每个位置都对应一个数字，则由1所对应的数字所产生的就是一个集合，例如：

000	{}
001	{3}
010	{2}
011	{2,3}
100	{1}
101	{1,3}
110	{1,2}
111	{1,2,3}

了解这个方法之后，剩下的就是如何产生二进位数？有许多方法可以使用，您可以使用unsigned型别加上&位元运算来产生，这边则是使用阵列搜 寻，首先阵列内容全为0，找第一个1，在还没找到之前将走访过的内容变为0，而第一个找到的0则变为1，如此重复直到所有的阵列元素都变为1为止，例如：

000 => 100 => 010 => 110 => 001 => 101 => 011 => 111

如果要产生字典顺序，例如若有4个元素，则：

{} => {1} => {1, 2} => {1, 2, 3} => {1, 2, 3, 4} =>
{1, 2, 4} =>
{1, 3} => {1, 3, 4} =>

{1, 4} =>
{2} => {2, 3} => {2, 3, 4} =>
{2, 4} =>
{3} => {3, 4} =>
{4}

简单的说，如果有n个元素要产生可能的集合，当依序产生集合时，如果最后一个元素是n，而倒数第二个元素是m的话，例如：
{a b c d e n}

则下一个集合就是{a b c d e+1}，再依序加入后续的元素。

例如有四个元素，而当产生{1 2 3 4}集合时，则下一个集合就是{1 2 3+1}，也就是{1 2 4}，由于最后一个元素还是4，所以下一个集合就是{1 2+1}，也就是{1 3}，接下来再加入后续元素4，也就是{1 3 4}，由于又遇到元素4，所以下一个集合是{1 3+1}，也就是{1 4}。

实作

C（无字典顺序）

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

#define MAXSIZE 20

int main(void) {
    char digit[MAXSIZE];
    int i, j;
    int n;

    printf("输入集合个数：");
    scanf("%d", &n);

    for(i = 0; i < n; i++)
        digit[i] = '0';

    printf("\n{}"); // 空集合

    while(1) {
        // 找第一个0，并将找到前所经过的元素变为0
        for(i = 0; i < n && digit[i] == '1'; digit[i] = '0', i++);

        if(i == n)   // 找不到0
            break;
        else              // 将第一个找到的0变为1
            digit[i] = '1';

        // 找第一个1，并记录对应位置
        for(i = 0; i < n && digit[i] == '0'; i++);

        printf("\n{%d", i+1);
```

```
        for(j = i + 1; j < n; j++)
            if(digit[j] == '1')
                printf(",%d", j + 1);

        printf("}");
    }

    printf("\n");

    return 0;
}
```

C（字典顺序）

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

#define MAXSIZE 20

int main(void) {
    int set[MAXSIZE];
    int i, n, position = 0;

    printf("输入集合个数：");
    scanf("%d", &n);
    printf("\n{}");
    set[position] = 1;

    while(1) {
        printf("\n{%d", set[0]);   // 印第一个数
        for(i = 1; i <= position; i++)
            printf(",%d", set[i]);
        printf("}");

        if(set[position] < n) {   // 递增集合个数
            set[position+1] = set[position] + 1;
            position++;
        }
        else if(position != 0) {   // 如果不是第一个位置
            position--;            // 倒退
            set[position]++;   // 下一个集合尾数
        }
        else   // 已倒退至第一个位置
            break;
    }

    printf("\n");

    return 0;
}
```

30.m 元素集合的 n 个元素子集

说明

假设有个集合拥有m个元素，任意的从集合中取出n个元素，则这n个元素所形成的可能子集有那些？

解法

假设有5个元素的集点，取出3个元素的可能子集如下：
{1 2 3}、{1 2 4 }、{1 2 5}、{1 3 4}、{1 3 5}、{1 4 5}、{2 3 4}、{2 3 5}、{2 4 5}、{3 4 5}

这些子集已经使用字典顺序排列，如此才可以观察出一些规则：
如果最右一个元素小于m，则如同码表一样的不断加1
如果右边一位已至最大值，则加1的位置往左移
每次加1的位置往左移后，必须重新调整右边的元素为递减顺序

所以关键点就在于哪一个位置必须进行加1的动作，到底是最右一个位置要加1？还是其它的位置？

在实际撰写程式时，可以使用一个变数positon来记录加1的位置，position的初值设定为n-1，因为我们要使用阵列，而最右边的索引值为最大 的n-1，在position位置的值若小于m就不断加1，如果大于m了，position就减1，也就是往左移一个位置；由于位置左移后，右边的元素会 经过调整，所以我们必须检查最右边的元素是否小于m，如果是，则position调整回n-1，如果不是，则positon维持不变。

实作

C

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

#define MAX 20

int main(void) {
	int set[MAX];
	int m, n, position;
	int i;

	printf("输入集合个数  m：");
	scanf("%d", &m);
	printf("输入取出元素  n：");
	scanf("%d", &n);

	for(i = 0; i < n; i++)
		set[i] = i + 1;

	// 显示第一个集合
	for(i = 0; i < n; i++)
```

```
            printf("%d ", set[i]);
        putchar('\n');

        position = n - 1;

        while(1) {
            if(set[n-1] == m)
                position--;
            else
                position = n - 1;

            set[position]++;

            // 调整右边元素
            for(i = position + 1; i < n; i++)
                set[i] = set[i-1] + 1;

            for(i = 0; i < n; i++)
                printf("%d ", set[i]);
            putchar('\n');

            if(set[0] >= m - n + 1)
                break;
        }

        return 0;
}
```

31.数字拆解

说明

这个题目来自于 数字拆解，我将之改为C语言的版本，并加上说明。

题目是这样的：

3 = 2+1 = 1+1+1 所以3有三种拆法

4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1 共五种

5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1

共七种

依此类推，请问一个指定数字NUM的拆解方法个数有多少个？

解法

我们以上例中最后一个数字5的拆解为例，假设f(n)为数字n的可拆解方式个数，而f(x, y)为使用y以下的数字来拆解x的方法个数，则观察：

5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1

使用函式来表示的话：
f(5) = f(4, 1) + f(3,2) + f(2,3) + f(1,4) + f(0,5)

其中f(1, 4) = f(1, 3) + f(1, 2) + f(1, 1)，但是使用大于1的数字来拆解1没有意义，所以f(1, 4) = f(1, 1)，而同样的，f(0, 5)会等于f(0, 0)，所以：
f(5) = f(4, 1) + f(3,2) + f(2,3) + f(1,1) + f(0,0)

依照以上的说明，使用动态程式规画（Dynamic programming）来进行求解，其中f(4,1)其实就是f(5-1, min(5-1,1))，f(x, y)就等于f(n-y, min(n-x, y))，其中n为要拆解的数字，而min()表示取两者中较小的数。

使用一个二维阵列表格table[x][y]来表示f(x, y)，刚开始时，将每列的索引0与索引1元素值设定为1，因为任何数以0以下的数拆解必只有1种，而任何数以1以下的数拆解也必只有1种：
```
for(i = 0; i < NUM +1; i++){
    table[i][0] = 1; // 任何数以0以下的数拆解必只有1种
    table[i][1] = 1; // 任何数以1以下的数拆解必只有1种
}
```

接下来就开始一个一个进行拆解了，如果数字为NUM，则我们的阵列维度大小必须为NUM x (NUM/2+1)，以数字10为例，其维度为10 x 6我们的表格将会如下所示：
```
1 1 0 0 0 0
1 1 0 0 0 0
1 1 2 0 0 0
1 1 2 3 0 0
1 1 3 4 5 0
1 1 3 5 6 7
1 1 4 7 9 0
1 1 4 8 0 0
1 1 5 0 0 0
1 1 0 0 0 0
```

实作

C

```
#include <stdio.h>
#include <stdlib.h>
#define NUM 10      //  要拆解的数字
#define DEBUG 0

int main(void) {
    int table[NUM][NUM/2+1] = {0}; // 动态规画表格
    int count = 0;
    int result = 0;
    int i, j, k;

    printf("数字拆解\n");
    printf("3 = 2+1 = 1+1+1 所以3有三种拆法\n");
    printf("4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1");
    printf("共五种\n");
```

```
    printf("5 = 4 + 1 = 3 + 2 = 3 + 1 + 1");
    printf(" = 2 + 2 + 1 = 2 + 1 + 1 + 1 = 1 + 1 +1 +1 +1");
    printf("共七种\n");
    printf("依此类推，求 %d 有几种拆法？", NUM);

    // 初始化
    for(i = 0; i < NUM; i++){
        table[i][0] = 1;   // 任何数以0以下的数拆解必只有1种
        table[i][1] = 1;   // 任何数以1以下的数拆解必只有1种
    }

    // 动态规划
    for(i = 2; i <= NUM; i++){
      for(j = 2; j <= i; j++){
            if(i + j > NUM) // 大于 NUM
                continue;

            count = 0;
            for(k = 1 ; k <= j; k++){
                count += table[i-k][(i-k >= k) ? k : i-k];
            }
            table[i][j] = count;
        }
    }

    // 计算并显示结果
    for(k = 1 ; k <= NUM; k++)
        result += table[NUM-k][(NUM-k >= k) ? k : NUM-k];
    printf("\n\nresult: %d\n", result);

    if(DEBUG) {
        printf("\n除错资讯\n");
        for(i = 0; i < NUM; i++) {
            for(j = 0; j < NUM/2+1; j++)
                printf("%2d", table[i][j]);
            printf("\n");
        }
    }

    return 0;
}
```

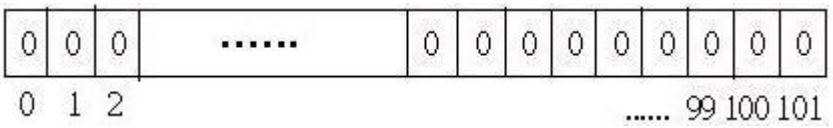
32.得分排行

说明假设有一教师依学生座号输入考试分数，现希望在输入完毕后自动显示学生分数的排行，当然学生的分数可能相同。

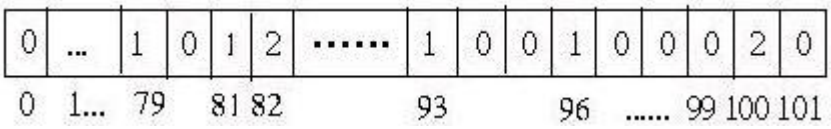
解法这个问题基本上要解不难，只要使用额外的一个排行阵列走访分数阵列就可以了，直接使用下面的程式片段作说明：

```
for(i = 0; i < count; i++) {
    juni[i] = 1;
    for(j = 0; j < count; j++) {
        if(score[j] > score[i])
            juni[i]++;
    }
}
printf("得分\t排行\n");
for(i = 0; i < count; i++)
    printf("%d\t%d\n", score[i], juni[i]);
```

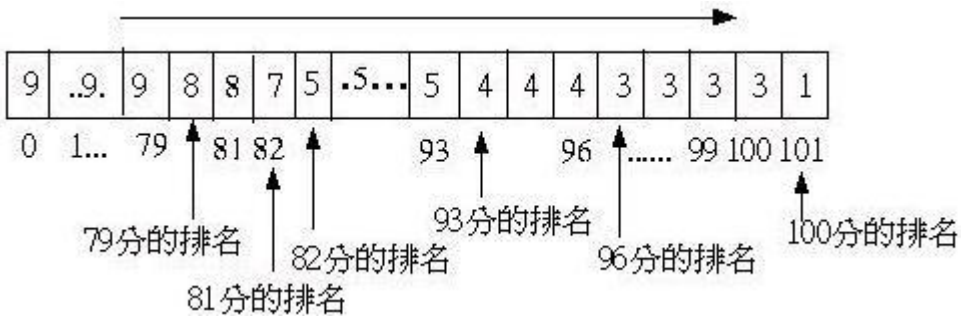
上面这个方法虽然简单，但是反覆计算的次数是n^2，如果n值变大，那么运算的时间就会拖长；改变juni阵列的长度为n+2，并将初始值设定为0，如下所示：



接下来走访分数阵列，并在分数所对应的排行阵列索引元素上加1，如下所示：



将排行阵列最右边的元素设定为1，然后依序将右边的元素值加至左边一个元素，最后排行阵列中的「分数+1」」就是得该分数的排行，如下所示：



这样的方式看起来复杂，其实不过在计算某分数之前排行的人数，假设89分之前的排行人数为x人，则89分自然就是x+1了，这也是为什么排行阵列最右边要设定为1的原因；如果89分有y人，则88分自然就是x+y+1，整个阵列右边元素向左加的原因正是如此。

如果分数有负分的情况，由于C/C++或Java等程式语言无法处理负的索引，所以必须加上一个偏移值，将所有的分数先往右偏移一个范围即可，最后显示的时候记得减回偏移值就可以了。

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
#define MIN 0

int main(void) {
    int score[MAX+1] = {0};
    int juni[MAX+2] = {0};
    int count = 0, i;

    do {
        printf("输入分数，-1结束：");
```

```
        scanf("%d", &score[count++]);
    } while(score[count-1] != -1);
    count--;

    for(i = 0; i < count; i++)
        juni[score[i]]++;
    juni[MAX+1] = 1;

    for(i = MAX; i >= MIN; i--)
        juni[i] = juni[i] + juni[i+1];
    printf("得分\t排行\n");
    for(i = 0; i < count; i++)
        printf("%d\t%d\n", score[i], juni[score[i]+1]);

    return 0;
}
```

33.选择、插入、气泡排序

说明选择排序（Selection sort）、插入排序（Insertion sort）与气泡排序（Bubble sort）这三个排序方式是初学排序所必须知道的三个基本排序方式，它们由于速度不快而不实用（平均与最快的时间复杂度都是$O(n^2)$），然而它们排序的方式确是值得观察与探讨的。

解法

选择排序

将要排序的对象分作两部份，一个是已排序的，一个是未排序的，从后端未排序部份选择一个最小值，并放入前端已排序部份的最后一个，例如：

排序前：70 80 31 37 10 1 48 60 33 80

[1] 80 31 37 10 70 48 60 33 80 选出最小值1
[1 10] 31 37 80 70 48 60 33 80 选出最小值10
[1 10 31] 37 80 70 48 60 33 80 选出最小值31
[1 10 31 33] 80 70 48 60 37 80 .....
[1 10 31 33 37] 70 48 60 80 80 .....
[1 10 31 33 37 48] 70 60 80 80 .....
[1 10 31 33 37 48 60] 70 80 80 .....
[1 10 31 33 37 48 60 70] 80 80 .....
[1 10 31 33 37 48 60 70 80] 80 .....

插入排序

像是玩朴克一样，我们将牌分作两堆，每次从后面一堆的牌抽出最前端的牌，然后插入前面一堆牌的适当位置，例如：

排序前：92 77 67 8 6 84 55 85 43 67

[77 92] 67 8 6 84 55 85 43 67 将77插入92前

[67 77 92] 8 6 84 55 85 43 67 将67插入77前
[8 67 77 92] 6 84 55 85 43 67 将8插入67前
[6 8 67 77 92] 84 55 85 43 67 将6插入8前
[6 8 67 77 84 92] 55 85 43 67 将84插入92前
[6 8 55 67 77 84 92] 85 43 67 将55插入67前
[6 8 55 67 77 84 85 92] 43 67 .....
[6 8 43 55 67 77 84 85 92] 67 .....
[6 8 43 55 67 67 77 84 85 92] .....

气泡排序法

顾名思义，就是排序时，最大的元素会如同气泡一样移至右端，其利用比较相邻元素的方法，将大的元素交换至右端，所以大的元素会不断的往右移动，直到适当的位置为止。

基本的气泡排序法可以利用旗标的方式稍微减少一些比较的时间，当寻访完阵列后都没有发生任何的交换动作，表示排序已经完成，而无需再进行之后的回圈比较与交换动作，例如：

排序前：95 27 90 49 80 58 6 9 18 50

27 90 49 80 58 6 9 18 50 [95] 95浮出
27 49 80 58 6 9 18 50 [90 95] 90浮出
27 49 58 6 9 18 50 [80 90 95] 80浮出
27 49 6 9 18 50 [58 80 90 95] .....
27 6 9 18 49 [50 58 80 90 95] .....
6 9 18 27 [49 50 58 80 90 95] .....
6 9 18 [27 49 50 58 80 90 95] 由于接下来不会再发生交换动作，排序提早结束

在上面的例子当中，还加入了一个观念，就是当进行至i与i+1时没有交换的动作，表示接下来的i+2至n已经排序完毕，这也增进了气泡排序的效率。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void selsort(int[]);    // 选择排序
void insort(int[]);     // 插入排序
void bubsort(int[]);    // 气泡排序

int main(void) {
    int number[MAX] = {0};
    int i;

    srand(time(NULL));

    printf("排序前：");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
```

```
        printf("%d ", number[i]);
    }

    printf("\n请选择排序方式：\n");
    printf("(1)选择排序\n(2)插入排序\n(3)气泡排序\n:");
    scanf("%d", &i);

    switch(i) {
        case 1:
            selsort(number); break;
        case 2:
            insort(number); break;
        case 3:
            bubsort(number); break;
        default:
            printf("选项错误(1..3)\n");
    }

    return 0;
}

void selsort(int number[]) {
    int i, j, k, m;

    for(i = 0; i < MAX-1; i++) {
        m = i;
        for(j = i+1; j < MAX; j++)
            if(number[j] < number[m])
                m = j;

        if( i != m)
            SWAP(number[i], number[m])

        printf("第 %d 次排序：", i+1);
        for(k = 0; k < MAX; k++)
            printf("%d ", number[k]);
        printf("\n");
    }
 }

 void insort(int number[]) {
    int i, j, k, tmp;

    for(j = 1; j < MAX; j++) {
        tmp = number[j];
        i = j - 1;
        while(tmp < number[i]) {
            number[i+1] = number[i];
            i--;
            if(i == -1)
                break;
```

```
            }
            number[i+1] = tmp;

            printf("第 %d 次排序：", j);
            for(k = 0; k < MAX; k++)
                printf("%d ", number[k]);
            printf("\n");
        }
}

void bubsort(int number[]) {
        int i, j, k, flag = 1;

        for(i = 0; i < MAX-1 && flag == 1; i++) {
            flag = 0;
            for(j = 0; j < MAX-i-1; j++) {
                if(number[j+1] < number[j]) {
                    SWAP(number[j+1], number[j]);
                    flag = 1;
                }
            }

            printf("第 %d 次排序：", i+1);
            for(k = 0; k < MAX; k++)
                printf("%d ", number[k]);
            printf("\n");
        }
}
```

34.Shell 排序法 - 改良的插入排序

说明

插入排序法由未排序的后半部前端取出一个值，插入已排序前半部的适当位置，概念简单但速度不快。

排序要加快的基本原则之一，是让后一次的排序进行时，尽量利用前一次排序后的结果，以加快排序的速度，Shell排序法即是基于此一概念来改良插入排序法。

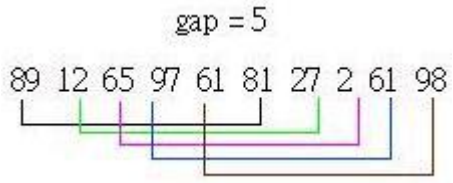
解法

Shell排序法最初是D.L Shell于1959所提出，假设要排序的元素有n个，则每次进行插入排序时并不是所有的元素同时进行时，而是取一段间隔。

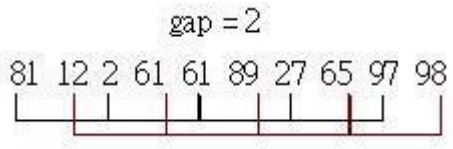
Shell首先将间隔设定为n/2，然后跳跃进行插入排序，再来将间隔n/4，跳跃进行排序动作，再来间隔设定为n/8、n/16，直到间隔为1之后的最 后一次排序终止，由于上一次的排序动作都会将固定间隔内的元素排序好，所以当间隔越来越小时，某些元素位于正确位置的机率越高，因此最后几次的排序动作将 可以大幅减低。

举个例子来说，假设有一未排序的数字如右：89 12 65 97 61 81 27 2 61 98

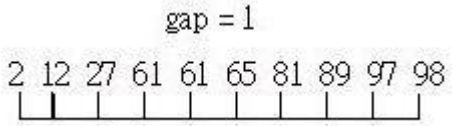
数字的总数共有10个，所以第一次我们将间隔设定为10 / 2 = 5，此时我们对间隔为5的数字进行排序，如下所示：



画线连结的部份表示 要一起进行排序的部份，再来将间隔设定为5 / 2的商，也就是2，则第二次的插入排序对象如下所示：



再来间隔设定为2 / 2 = 1，此时就是单纯的插入排序了，由于大部份的元素都已大致排序过了，所以最后一次的插入排序几乎没作什么排序动作了：



将间隔设定为n / 2是D.L Shell最初所提出，在教科书中使用这个间隔比较好说明，然而Shell排序法的关键在于间隔的选定，例如Sedgewick证明选用以下的间隔可以加 快Shell排序法的速度：

$$4*(2^j)^2 + 3*(2^j) + 1$$

$$j = \log_2 [\frac{-3 + \sqrt{16*n - 7}}{8}]$$

其中$4*(2^j)^2 + 3*(2^j) + 1$不可超过元素总数n值，使用上式找出j后代入$4*(2^j)^2 + 3*(2^j) + 1$求得第一个间隔，然后将2^j除以2代入求得第二个间隔，再来依此类推。

后来还有人证明有其它的间隔选定法可以将Shell排序法的速度再加快；另外Shell排序法的概念也可以用来改良气泡排序法。

实作

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void shellsort(int[]);

int main(void) {
    int number[MAX] = {0};
    int i;

    srand(time(NULL));
```

```
    printf("排序前： ");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    shellsort(number);

    return 0;
}

void shellsort(int number[]) {
    int i, j, k, gap, t;

    gap = MAX / 2;

    while(gap > 0) {
        for(k = 0; k < gap; k++) {
            for(i = k+gap; i < MAX; i+=gap) {
                for(j = i - gap; j >= k; j-=gap) {
                    if(number[j] > number[j+gap]) {
                        SWAP(number[j], number[j+gap]);
                    }
                    else
                        break;
                }
            }
        }

        printf("\ngap = %d： ", gap);
        for(i = 0; i < MAX; i++)
            printf("%d ", number[i]);
        printf("\n");

        gap /= 2;
    }
}
```

35.Shaker 排序法 - 改良的气泡排序

说明

请看看之前介绍过的气泡排序法：

```
    for(i = 0; i < MAX-1 && flag == 1; i++) {
        flag = 0;
        for(j = 0; j < MAX-i-1; j++) {
```

```
            if(number[j+1] < number[j]) {
                SWAP(number[j+1], number[j]);
                flag = 1;
            }
        }
    }
```

事实上这个气泡排序法已经不是单纯的气泡排序了，它使用了旗标与右端左移两个方法来改进排序的效能，而Shaker排序法使用到后面这个观念进一步改良气泡排序法。

解法

在上面的气泡排序法中，交换的动作并不会一直进行至阵列的最后一个，而是会进行至MAX-i-1，所以排序的过程中，阵列右方排序好的元素会一直增加，使得左边排序的次数逐渐减少，如我们的例子所示：

排序前：95 27 90 49 80 58 6 9 18 50

27 90 49 80 58 6 9 18 50 [95] 95浮出
27 49 80 58 6 9 18 50 [90 95] 90浮出
27 49 58 6 9 18 50 [80 90 95] 80浮出
27 49 6 9 18 50 [58 80 90 95] .....
27 6 9 18 49 [50 58 80 90 95] .....
6 9 18 27 [49 50 58 80 90 95] .....
6 9 18 [27 49 50 58 80 90 95]

方括号括住的部份表示已排序完毕，Shaker排序使用了这个概念，如果让左边的元素也具有这样的性质，让左右两边的元素都能先排序完成，如此未排序的元素会集中在中间，由于左右两边同时排序，中间未排序的部份将会很快的减少。

方法就在于气泡排序的双向进行，先让气泡排序由左向右进行，再来让气泡排序由右往左进行，如此完成一次排序的动作，而您必须使用left与right两个旗标来记录左右两端已排序的元素位置。

一个排序的例子如下所示：

排序前：45 19 77 81 13 28 18 19 77 11

往右排序：19 45 77 13 28 18 19 77 11 [81]
向左排序：[11] 19 45 77 13 28 18 19 77 [81]

往右排序：[11] 19 45 13 28 18 19 [77 77 81]
向左排序：[11 13] 19 45 18 28 19 [77 77 81]

往右排序：[11 13] 19 18 28 19 [45 77 77 81]
向左排序：[11 13 18] 19 19 28 [45 77 77 81]

往右排序：[11 13 18] 19 19 [28 45 77 77 81]
向左排序：[11 13 18 19 19] [28 45 77 77 81]

如上所示，括号中表示左右两边已排序完成的部份，当left > right时，则排序完成。

实作

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void shakersort(int[]);

int main(void) {
    int number[MAX] = {0};
    int i;

    srand(time(NULL));
```

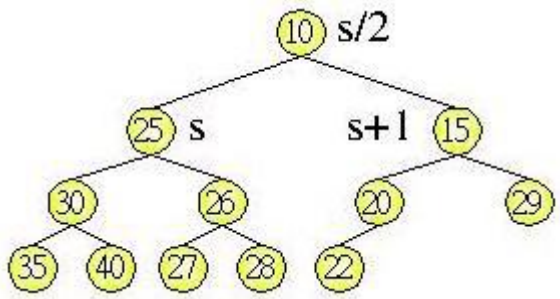
36.排序法 - 改良的选择排序

说明

选择排序法的概念简单，每次从未排序部份选一最小值，插入已排序部份的后端，其时间主要花费于在整个未排序部份寻找最小值，如果能让搜寻最小值的方式加 快，选择排序法的速率也就可以加快，Heap排序法让搜寻的路径由树根至最后一个树叶，而不是整个未排序部份，因而称之为改良的选择排序法。

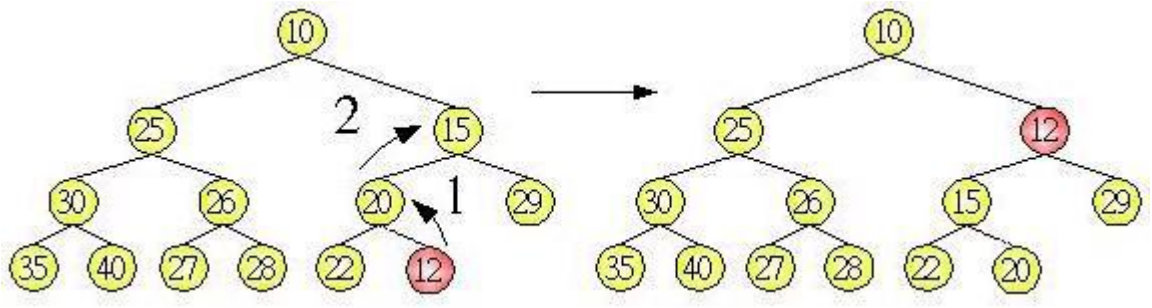
解法

Heap排序法使用Heap Tree（堆积树），树是一种资料结构，而堆积树是一个二元树，也就是每一个父节点最多只有两个子节点（关于树的详细定义还请见资料结构书籍），堆积树的 父节点若小于子节点，则称之为最小堆积（Min Heap），父节点若大于子节点，则称之为最大堆积（Max Heap），而同一层的子节点则无需理会其大小关系，例如下面就是一个堆积树：



可以使用一维阵列来储存堆积树的所有元素与其顺序，为了计算方便，使用的起始索引是1而不是0，索引1是树根位置，如果左子节点储存在阵列中的索引为s，则其父节点的索引为s/2，而右子节点为s+1，就如上图所示，将上图的堆积树转换为一维阵列之后如下所示：

首先必须知道如何建立堆积树，加至堆积树的元素会先放置在最后一个树叶节点位置，然后检查父节点是否小于子节点（最小堆积），将小的元素不断与父节点交换，直到满足堆积树的条件为止，例如在上图的堆积加入一个元素12，则堆积树的调整方式如下所示：

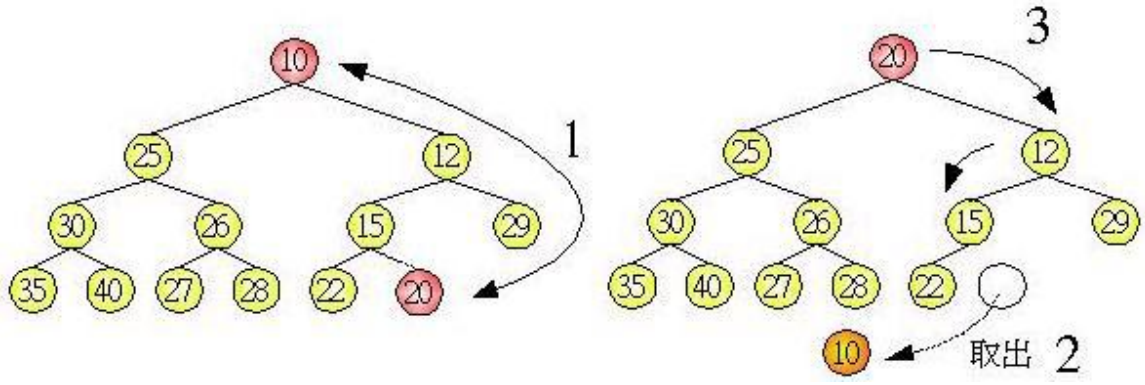


建立好堆积树之后，树根一定是所有元素的最小值，您的目的就是：

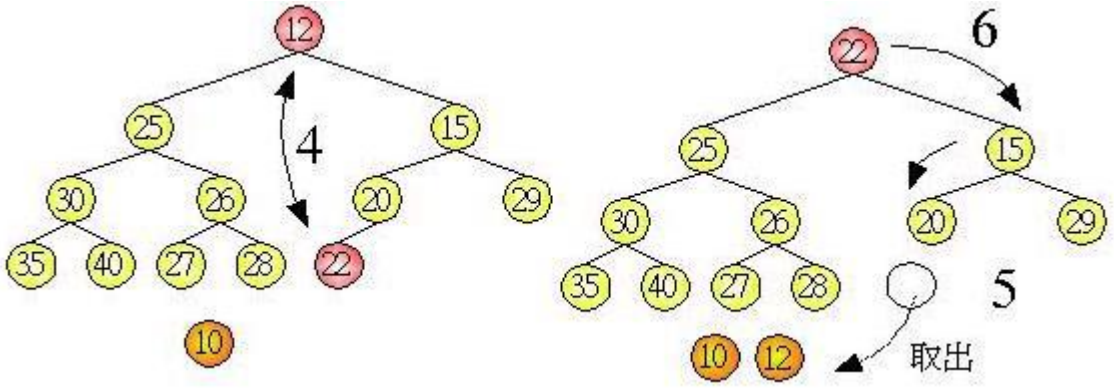
将最小值取出

然后调整树为堆积树

不断重复以上的步骤，就可以达到排序的效果，最小值的取出方式是将树根与最后一个树叶节点交换，然后切下树叶节点，重新调整树为堆积树，如下所示：



调整完毕后，树根节点又是最小值了，于是我们可以重覆这个步骤，再取出最小值，并调整树为堆积树，如下所示：



如此重覆步骤之后，由于使用一维阵列来储存堆积树，每一次将树叶与树根交换的动作就是将最小值放至后端的阵列，所以最后阵列就是变为已排序的状态。

其实堆积在调整的过程中，就是一个选择的行为，每次将最小值选至树根，而选择的路径并不是所有的元素，而是由树根至树叶的路径，因而可以加快选择的过程， 所以Heap排序法才会被称之为改良的选择排序法。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void createheap(int[]);
void heapsort(int[]);

int main(void) {
	int number[MAX+1] = {-1};
	int i, num;

	srand(time(NULL));

	printf("排序前： ");
	for(i = 1; i <= MAX; i++) {
		number[i] = rand() % 100;
		printf("%d ", number[i]);
	}

	printf("\n建立堆积树： ");
	createheap(number);
	for(i = 1; i <= MAX; i++)
		printf("%d ", number[i]);
	printf("\n");

	heapsort(number);

	printf("\n");

	return 0;
}

void createheap(int number[]) {
	int i, s, p;
	int heap[MAX+1] = {-1};

	for(i = 1; i <= MAX; i++) {
		heap[i] = number[i];
		s = i;
		p = i / 2;
		while(s >= 2 && heap[p] > heap[s]) {
			SWAP(heap[p], heap[s]);
			s = p;
			p = s / 2;
		}
	}

	for(i = 1; i <= MAX; i++)
		number[i] = heap[i];

}
```

```
void heapsort(int number[]) {
    int i, m, p, s;

    m = MAX;
    while(m > 1) {
        SWAP(number[1], number[m]);
        m--;

        p = 1;
        s = 2 * p;

        while(s <= m) {
            if(s < m && number[s+1] < number[s])
                s++;
            if(number[p] <= number[s])
                break;
            SWAP(number[p], number[s]);
            p = s;
            s = 2 * p;
        }

        printf("\n排序中：");
        for(i = MAX; i > 0; i--)
            printf("%d ", number[i]);
    }
}
```

37.快速排序法（一）

说明快速排序法（quick sort）是目前所公认最快的排序方法之一（视解题的对象而定），虽然快速排序法在最差状况下可以达$O(n^2)$，但是在多数的情况下，快速排序法的效率表现是相当不错的。

快速排序法的基本精神是在数列中找出适当的轴心，然后将数列一分为二，分别对左边与右边数列进行排序，而影响快速排序法效率的正是轴心的选择。

这边所介绍的第一个快速排序法版本，是在多数的教科书上所提及的版本，因为它最容易理解，也最符合轴心分割与左右进行排序的概念，适合对初学者进行讲解。

解法这边所介绍的快速演算如下：将最左边的数设定为轴，并记录其值为 s

迴圈处理：

令索引 i 从数列左方往右方找，直到找到大于 s 的数

令索引 j 从数列左右方往左方找，直到找到小于 s 的数

如果 i >= j，则离开回圈

如果 i < j，则交换索引i与j两处的值

将左侧的轴与 j 进行交换

对轴左边进行递回

对轴右边进行递回

透过以下演算法，则轴左边的值都会小于s，轴右边的值都会大于s，如此再对轴左右两边进行递回，就可以对完成排序的目的，例如下面的实例，*表示要交换的数，[]表示轴：

```
  [41]   24   76*   11   45   64   21   69   19   36*
  [41]   24   36   11   45*   64   21   69   19*   76
  [41]   24   36   11   19   64*   21*   69   45   76
  [41]   24   36   11   19   21   64   69   45   76
  21   24   36   11   19   [41]   64   69   45   76
```

在上面的例子中，41左边的值都比它小，而右边的值都比它大，如此左右再进行递回至排序完成。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);

int main(void) {
	int number[MAX] = {0};
	int i, num;

	srand(time(NULL));

	printf("排序前： ");
	for(i = 0; i < MAX; i++) {
		number[i] = rand() % 100;
		printf("%d ", number[i]);
	}

	quicksort(number, 0, MAX-1);

	printf("\n排序后： ");
	for(i = 0; i < MAX; i++)
		printf("%d ", number[i]);

	printf("\n");

	return 0;
}

void quicksort(int number[], int left, int right) {
	int i, j, s;

	if(left < right) {
		s = number[left];
		i = left;
		j = right + 1;
```

```
        while(1) {
            // 向右找
            while(i + 1 < number.length && number[++i] < s) ;
            // 向左找
            while(j -1 > -1 && number[--j] > s) ;
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        number[left] = number[j];
        number[j] = s;

        quicksort(number, left, j-1);     // 对左边进行递回
        quicksort(number, j+1, right);    // 对右边进行递回
    }
}
```

38.快速排序法（二）

说明在快速排序法（一）中，每次将最左边的元素设为轴，而之前曾经说过，快速排序法的加速在于轴的选择，在这个例子中，只将轴设定为中间的元素，依这个元素作基准进行比较，这可以增加快速排序法的效率。

解法在这个例子中，取中间的元素s作比较，同样的先得右找比s大的索引 i，然后找比s小的索引 j，只要两边的索引还没有交会，就交换 i 与 j 的元素值，这次不用再进行轴的交换了，因为在寻找交换的过程中，轴位置的元素也会参与交换的动作，例如：

```
41  24  76  11  45  64  21  69  19  36
```

首先left为0，right为9，(left+right)/2 = 4（取整数的商），所以轴为索引4的位置，比较的元素是45，您往右找比45大的，往左找比45小的进行交换：

```
 41  24  76*  11  [45]  64  21  69  19  *36
 41  24  36  11  45*  64  21  69  19*  76
 41  24  36  11  19  64*  21*  69  45  76
 [41  24  36  11  19  21]  [64  69  45  76]
```

完成以上之后，再初别对左边括号与右边括号的部份进行递回，如此就可以完成排序的目的。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);
```

```
int main(void) {
    int number[MAX] = {0};
    int i, num;
    srand(time(NULL));
    printf("排序前： ");
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
        printf("%d ", number[i]);
    }

    quicksort(number, 0, MAX-1);
    printf("\n排序后： ");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n");
    return 0;
}

void quicksort(int number[], int left, int right) {
    int i, j, s;
    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;  // 向右找
            while(number[--j] > s) ;  // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);    // 对左边进行递回
        quicksort(number, j+1, right);   // 对右边进行递回
    }
}
```

39.快速排序法（三）

说明

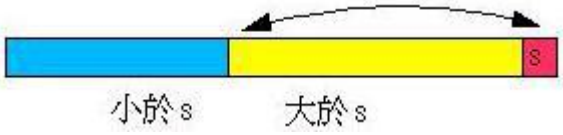
之前说过轴的选择是快速排序法的效率关键之一，在这边的快速排序法的轴选择方式更加快了快速排序法的效率，它是来自演算法名书 Introduction to Algorithms 之中。

解法

先说明这个快速排序法的概念，它以最右边的值s作比较的标准，将整个数列分为三个部份，一个是小于s的部份，一个是大

于s的部份，一个是未处理的部份，如下所示 ：

在排序的过程中，i 与 j 都会不断的往右进行比较与交换，最后数列会变为以下的状态：

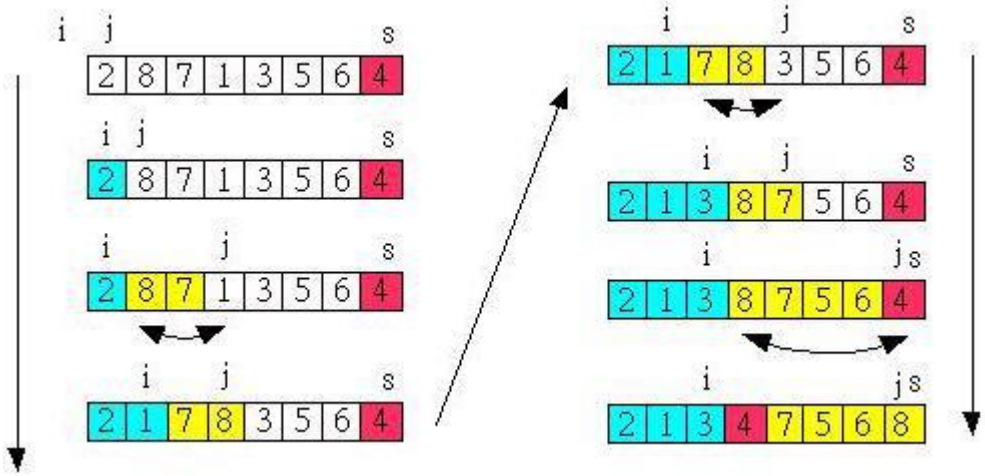


然后将s的值置于中间，接下来就以相同的步骤会左右两边的数列进行排序的动作，如下所示：

整个演算的过程，直接摘录书中的虚拟码来作说明：

```
QUICKSORT(A, p, r)
      if p < r
            then q <- PARTITION(A, p, r)
                        QUICKSORT(A, p, q-1)
                        QUICKSORT(A, q+1, r)
end QUICKSORT

PARTITION(A, p, r)
      x <- A[r]
      i <- p-1
      for j <- p to r-1
            do if A[j] <= x
                        then    i <- i+1
                                    exchange A[i]<->A[j]
      exchange A[i+1]<->A[r]
      return i+1
end PARTITION
```

一个实际例子的演算如下所示：



```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int partition(int[], int, int);
void quicksort(int[], int, int);

int main(void) {
	int number[MAX] = {0};
	int i, num;

	srand(time(NULL));

	printf("排序前： ");
	for(i = 0; i < MAX; i++) {
		number[i] = rand() % 100;
		printf("%d ", number[i]);
	}

	quicksort(number, 0, MAX-1);

	printf("\n排序后： ");
	for(i = 0; i < MAX; i++)
		printf("%d ", number[i]);

	printf("\n");

	return 0;
}

int partition(int number[], int left, int right) {
	int i, j, s;

	s = number[right];
	i = left - 1;

	for(j = left; j < right; j++) {
		if(number[j] <= s) {
			i++;
			SWAP(number[i], number[j]);
		}
	}

	SWAP(number[i+1], number[right]);
	return i+1;
}
```

```
void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

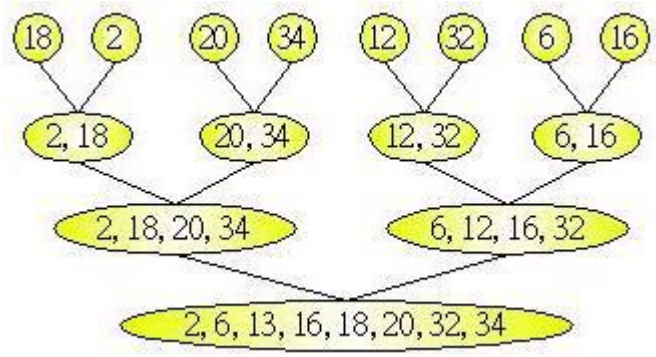
40.合并排序法

说明之前所介绍的排序法都是在同一个阵列中的排序，考虑今日有两笔或两笔以上的资料，它可能是不同阵列中的资料，或是不同档案中的资料，如何为它们进行排序？

解法可以使用合并排序法，合并排序法基本是将两笔已排序的资料合并并进行排序，如果所读入的资料尚未排序，可以先利用其它的排序方式来处理这两笔资料，然后再将排序好的这两笔资料合并。

有人问道，如果两笔资料本身就无排序顺序，何不将所有的资料读入，再一次进行排序？排序的精神是尽量利用资料已排序的部份，来加快排序的效率，小笔资料的 排序较为快速，如果小笔资料排序完成之后，再合并处理时，因为两笔资料都有排序了，所有在合并排序时会比单纯读入所有的资料再一次排序来的有效率。

那么可不可以直接使用合并排序法本身来处理整个排序的动作？而不动用到其它的排序方式？答案是肯定的，只要将所有的数字不断的分为两个等分，直到最后剩一个数字为止，然后再反过来不断的合并，就如下图所示：



不过基本上分割又会花去额外的时间，不如使用其它较好的排序法来排序小笔资料，再使用合并排序来的有效率。
下面这个程式范例，我们使用快速排序法来处理小笔资料排序，然后再使用合并排序法处理合并的动作。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX1 10
#define MAX2 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int partition(int[], int, int);
void quicksort(int[], int, int);
void mergesort(int[], int, int[], int, int[]);

int main(void) {
```

```
    int number1[MAX1] = {0};
    int number2[MAX1] = {0};
    int number3[MAX1+MAX2] = {0};
    int i, num;

    srand(time(NULL));
    printf("排序前： ");
    printf("\nnumber1[]： ");
    for(i = 0; i < MAX1; i++) {
        number1[i] = rand() % 100;
        printf("%d ", number1[i]);
    }

    printf("\nnumber2[]： ");
    for(i = 0; i < MAX2; i++) {
        number2[i] = rand() % 100;
        printf("%d ", number2[i]);
    }

    // 先排序两笔资料
    quicksort(number1, 0, MAX1-1);
    quicksort(number2, 0, MAX2-1);
    printf("\n排序后： ");
    printf("\nnumber1[]： ");
    for(i = 0; i < MAX1; i++)
        printf("%d ", number1[i]);
    printf("\nnumber2[]： ");
    for(i = 0; i < MAX2; i++)
        printf("%d ", number2[i]);

    // 合并排序
    mergesort(number1, MAX1, number2, MAX2, number3);
    printf("\n合并后： ");
    for(i = 0; i < MAX1+MAX2; i++)
        printf("%d ", number3[i]);

    printf("\n");
    return 0;
}

int partition(int number[], int left, int right) {
    int i, j, s;
    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
    }
```

```
    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;
    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}

void mergesort(int number1[], int M, int number2[], int N, int number3[]) {
    int i = 0, j = 0, k = 0;

    while(i < M && j < N) {
        if(number1[i] <= number2[j])
            number3[k++] = number1[i++];
        else
            number3[k++] = number2[j++];
    }
    while(i < M)
        number3[k++] = number1[i++];
    while(j < N)
        number3[k++] = number2[j++];
}
```

41.基数排序法

说明在之前所介绍过的排序方法，都是属于「比较性」的排序法，也就是每次排序时 ，都是比较整个键值的大小以进行排序。

这边所要介绍的「基数排序法」（radix sort）则是属于「分配式排序」（distribution sort），基数排序法又称「桶子法」（bucket sort）或bin sort，顾名思义，它是透过键值的部份资讯，将要排序的元素分配至某些「桶」中，藉以达到排序的作用，基数排序法是属于稳定性的排序，其时间复杂度为O (nlog(r)m)，其中r为所采取的基数，而m为堆数，在某些时候，基数排序法的效率高于其它的比较性排序法。

解法基数排序的方式可以采用LSD（Least sgnificant digital）或MSD（Most sgnificant digital），LSD的排序方式由键值的最右边开始，而MSD则相反，由键值的最左边开始。

以LSD为例，假设原来有一串数值如下所示：

73，22，93，43，55，14，28，65，39，81

首先根据个位数的数值，在走访数值时将它们分配至编号0到9的桶子中：

0	1	2	3	4	5	6	7	8	9
	81				65				39
			43	14	55			28	
			93						

		22	73						

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

81，22，73，93，43，14，55，65，28，39

接着再进行一次分配，这次是根据十位数来分配：

0	1	2	3	4	5	6	7	8	9
	28	39							
	14	22		43	55	65	73	81	93

接下来将这些桶子中的数值重新串接起来，成为以下的数列：

14，22，28，39，43，55，65，73，81，93

这时候整个数列已经排序完毕；如果排序的对象有三位数以上，则持续进行以上的动作直至最高位数为止。

LSD的基数排序适用于位数小的数列，如果位数多的话，使用MSD的效率会比较好，MSD的方式恰与LSD相反，是由高位数为基底开始进行分配，其他的演 算方式则都相同。

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

int main(void) {
    int data[10] = {73, 22, 93, 43, 55, 14, 28, 65, 39, 81};
    int temp[10][10] = {0};
    int order[10] = {0};
    int i, j, k, n, lsd;
    k = 0;
    n = 1;
    printf("\n排序前: ");
    for(i = 0; i < 10; i++)
        printf("%d ", data[i]);
    putchar('\n');
    while(n <= 10) {
        for(i = 0; i < 10; i++) {
            lsd = ((data[i] / n) % 10);
            temp[lsd][order[lsd]] = data[i];
            order[lsd]++;
        }
        printf("\n重新排列: ");
        for(i = 0; i < 10; i++) {
            if(order[i] != 0)
                for(j = 0; j < order[i]; j++) {
                    data[k] = temp[i][j];
                    printf("%d ", data[k]);
                    k++;
                }
            order[i] = 0;
        }
        n *= 10;
```

```
            k = 0;
        }

        putchar('\n');
        printf("\n排序后: ");
        for(i = 0; i < 10; i++)
            printf("%d ", data[i]);
        return 0;
}
```

42.循序搜寻法（使用卫兵）

说明

搜寻的目的，是在「已排序的资料」中寻找指定的资料，而当中循序搜寻是最基本的搜寻法，只要从资料开头寻找到最后，看看是否找到资料即可。

解法

初学者看到循序搜寻，多数都会使用以下的方式来进行搜寻：

```
while(i < MAX) {
    if(number[i] == k) {
        printf("找到指定值");
        break;
    }
    i++;
}
```

这个方法基本上没有错，但是可以加以改善，可以利用设定卫兵的方式，省去if判断式，卫兵通常设定在数列最后或是最前方，假设设定在列前方好了（索引0的 位置），我们从数列后方向前找，如果找到指定的资料时，其索引值不是0，表示在数列走访完之前就找到了，在程式的撰写上，只要使用一个while回圈就可 以了。

下面的程式为了配合卫兵的设置，自行使用快速排序法先将产生的数列排序，然后才进行搜寻，若只是数字的话，通常您可以使用程式语言函式库所提供的搜寻函式。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

int search(int[]);
int partition(int[], int, int);
void quicksort(int[], int, int);

int main(void) {
    int number[MAX+1] = {0};
```

```
    int i, find;

    srand(time(NULL));

    for(i = 1; i <= MAX; i++)
        number[i] = rand() % 100;

    quicksort(number, 1, MAX);

    printf("数列： ");
    for(i = 1; i <= MAX; i++)
        printf("%d ", number[i]);

    printf("\n输入搜寻值： ");
    scanf("%d", &number[0]);

    if(find = search(number))
        printf("\n找到数值于索引 %d ", find);
    else
        printf("\n找不到数值");

    printf("\n");

    return 0;
}

int search(int number[]) {
    int i, k;

    k = number[0];
    i = MAX;

    while(number[i] != k)
        i--;

    return i;
}

int partition(int number[], int left, int right) {
    int i, j, s;

    s = number[right];
    i = left - 1;

    for(j = left; j < right; j++) {
        if(number[j] <= s) {
            i++;
            SWAP(number[i], number[j]);
        }
    }
```

```
    SWAP(number[i+1], number[right]);
    return i+1;
}

void quicksort(int number[], int left, int right) {
    int q;

    if(left < right) {
        q = partition(number, left, right);
        quicksort(number, left, q-1);
        quicksort(number, q+1, right);
    }
}
```

43.二分搜寻法（搜寻原则的代表）

说明如果搜寻的数列已经有排序，应该尽量利用它们已排序的特性，以减少搜寻比对的次数，这是搜寻的基本原则，二分搜寻法是这个基本原则的代表。

解法在二分搜寻法中，从数列的中间开始搜寻，如果这个数小于我们所搜寻的数，由于数列已排序，则该数左边的数一定都小于要搜寻的对象，所以无需浪费时间在左边的数；如果搜寻的数大于所搜寻的对象，则右边的数无需再搜寻，直接搜寻左边的数。

所以在二分搜寻法中，将数列不断的分为两个部份，每次从分割的部份中取中间数比对，例如要搜寻92于以下的数列，首先中间数索引为(0+9)/2 = 4（索引由0开始）：

[3 24 57 57 **67** 68 83 90 92 95]

由于67小于92，所以转搜寻右边的数列：

3 24 57 57 67 [68 83 **90** 92 95]

由于90小于92，再搜寻右边的数列，这次就找到所要的数了：

3 24 57 57 67 68 83 90 [**92** 95]

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);
int bisearch(int[], int);

int main(void) {
    int number[MAX] = {0};
    int i, find;
    srand(time(NULL));
```

```
    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
    }

    quicksort(number, 0, MAX-1);

    printf("数列：");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n输入寻找对象：");
    scanf("%d", &find);

    if((i = bisearch(number, find)) >= 0)
        printf("找到数字于索引  %d ", i);
    else
        printf("\n找不到指定数");

    printf("\n");

    return 0;
}

int bisearch(int number[], int find) {
    int low, mid, upper;

    low = 0;
    upper = MAX - 1;

    while(low <= upper) {
        mid = (low+upper) / 2;
        if(number[mid] < find)
            low = mid+1;
        else if(number[mid] > find)
            upper = mid - 1;
        else
            return mid;
    }

    return -1;
}

void quicksort(int number[], int left, int right) {
    int i, j, k, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
```

```
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 对左边进行递回
        quicksort(number, j+1, right);    // 对右边进行递回
    }
}
```

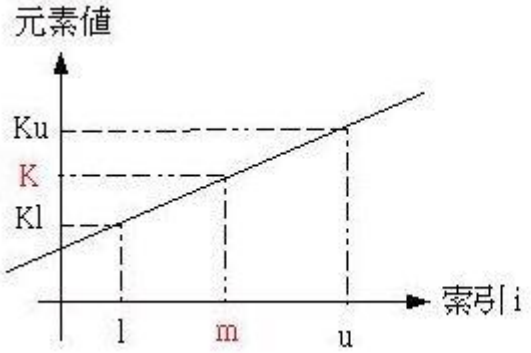
44.插补搜寻法

说明

如果却搜寻的资料分布平均的话，可以使用插补（Interpolation）搜寻法来进行搜寻，在搜寻的对象大于500时，插补搜寻法会比 二分搜寻法 来的快速。

解法

插补搜寻法是以资料分布的近似直线来作比例运算，以求出中间的索引并进行资料比对，如果取出的值小于要寻找的值，则提高下界，如果取出的值大于要寻找的 值，则降低下界，如此不断的减少搜寻的范围，所以其本原则与二分搜寻法是相同的，至于中间值的寻找是透过比例运算，如下所示，其中K是指定要寻找的对象， 而m则是可能的索引值：



$$\frac{K-Kl}{Ku-Kl} = \frac{m-l}{u-l}$$

$$m = (u-l)* \frac{K-Kl}{Ku-Kl} + l$$

实作

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void quicksort(int[], int, int);
int intsrch(int[], int);

int main(void) {
```

```
    int number[MAX] = {0};
    int i, find;

    srand(time(NULL));

    for(i = 0; i < MAX; i++) {
        number[i] = rand() % 100;
    }

    quicksort(number, 0, MAX-1);

    printf("数列： ");
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);

    printf("\n输入寻找对象： ");
    scanf("%d", &find);

    if((i = intsrch(number, find)) >= 0)
        printf("找到数字于索引 %d ", i);
    else
        printf("\n找不到指定数");

    printf("\n");

    return 0;
}

int intsrch(int number[], int find) {
    int low, mid, upper;

    low = 0;
    upper = MAX - 1;

    while(low <= upper) {
        mid = (upper-low)*
            (find-number[low])/(number[upper]-number[low])
            + low;
        if(mid < low || mid > upper)
            return -1;

        if(find < number[mid])
            upper = mid - 1;
        else if(find > number[mid])
            low = mid + 1;
        else
            return mid;
     }

     return -1;
}
```

```
void quicksort(int number[], int left, int right) {
    int i, j, k, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 对左边进行递回
        quicksort(number, j+1, right);    // 对右边进行递回
    }
}
```

45.费氏搜寻法

说明

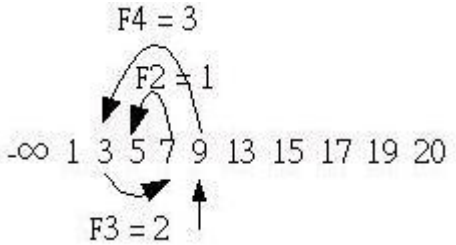
二分搜寻法每次搜寻时，都会将搜寻区间分为一半，所以其搜寻时间为O(log(2)n)，log(2)表示以2为底的log值，这边要介绍的费氏搜寻，其利用费氏数列作为间隔来搜寻下一个数，所以区间收敛的速度更快，搜寻时间为O(logn)。

解法

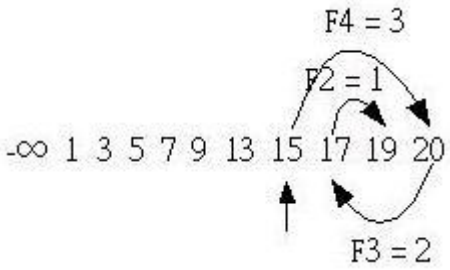
费氏搜寻使用费氏数列来决定下一个数的搜寻位置，所以必须先制作费氏数列，这在之前有提过；费氏搜寻会先透过公式计算求出第一个要搜寻数的位置，以及其代 表的费氏数，以搜寻对象10个数字来说，第一个费氏数经计算后一定是F5，而第一个要搜寻的位置有两个可能，例如若在下面的数列搜寻的话（为了计算方便， 通常会将索引0订作无限小的数，而数列由索引1开始）：

```
-infin; 1 3 5 7 9 13 15 17 19 20
```

如果要搜寻5的话，则由索引F5 = 5开始搜寻，接下来如果数列中的数小于指定搜寻值时，就往左找，大于时就向右，每次找的间隔是F4、F3、F2来寻找，当费氏数为0时还没找到，就表示寻找失败，如下所示：



由于第一个搜寻值索引F5 = 5处的值小于19，所以此时必须对齐数列右方，也就是将第一个搜寻值的索引改为F5+2 = 7，然后如同上述的方式进行搜寻，如下所示：



至于第一个搜寻值是如何找到的？我们可以由以下这个公式来求得，其中n为搜寻对象的个数：

$F_x + m = n$
$F_x <= n$

也就是说Fx必须找到不大于n的费氏数，以10个搜寻对象来说：
$F_x + m = 10$

取$F_x = 8$, m = 2，所以我们可以对照费氏数列得x = 6，然而第一个数的可能位置之一并不是F6，而是第x-1的费氏数，也就是$F_5 = 5$。

如果数列number在索引5处的值小于指定的搜寻值，则第一个搜寻位置就是索引5的位置，如果大于指定的搜寻值，则第一个搜寻位置必须加上m，也就是$F_5 + m = 5 + 2 = 7$，也就是索引7的位置，其实加上m的原因，是为了要让下一个搜寻值刚好是数列的最后一个位置。

费氏搜寻看来难懂，但只要掌握$F_x + m = n$这个公式，自己找几个实例算一次，很容易就可以理解；费氏搜寻除了收敛快速之外，由于其本身只会使用到加法与减法，在运算上也可以加快。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 15
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void createfib(void);        // 建立费氏数列
int findx(int, int);         // 找x值
int fibsearch(int[], int);   // 费氏搜寻
void quicksort(int[], int, int);   // 快速排序

int Fib[MAX] = {-999};

int main(void) {
    int number[MAX] = {0};
    int i, find;

    srand(time(NULL));

    for(i = 1; i <= MAX; i++) {
        number[i] = rand() % 100;
```

```
    }

    quicksort(number, 1, MAX);

    printf("数列：");
    for(i = 1; i <= MAX; i++)
        printf("%d ", number[i]);

    printf("\n输入寻找对象：");
    scanf("%d", &find);

    if((i = fibsearch(number, find)) >= 0)
        printf("找到数字于索引 %d ", i);
    else
        printf("\n找不到指定数");

    printf("\n");

    return 0;
}

// 建立费氏数列
void createfib(void) {
    int i;

    Fib[0] = 0;
    Fib[1] = 1;

    for(i = 2; i < MAX; i++)
        Fib[i] = Fib[i-1] + Fib[i-2];
}

// 找 x 值
int findx(int n, int find) {
    int i = 0;

    while(Fib[i] <= n)
        i++;

    i--;
    return i;
}

// 费式搜寻
int fibsearch(int number[], int find) {
    int i, x, m;

    createfib();

    x   = findx(MAX+1,find);
    m = MAX - Fib[x];
```

```
    printf("\nx = %d, m = %d, Fib[x] = %d\n\n",
                                        x, m, Fib[x]);

    x--;
    i = x;

    if(number[i] < find)
        i += m;

    while(Fib[x] > 0) {
        if(number[i] < find)
            i += Fib[--x];
        else if(number[i] > find)
            i -= Fib[--x];
        else
            return i;
    }
    return -1;
}

void quicksort(int number[], int left, int right) {
    int i, j, k, s;

    if(left < right) {
        s = number[(left+right)/2];
        i = left - 1;
        j = right + 1;

        while(1) {
            while(number[++i] < s) ;   // 向右找
            while(number[--j] > s) ;   // 向左找
            if(i >= j)
                break;
            SWAP(number[i], number[j]);
        }

        quicksort(number, left, i-1);     // 对左边进行递回
        quicksort(number, j+1, right);    // 对右边进行递回
    }
}
```

46.稀疏矩阵

说明

如果在矩阵中，多数的元素并没有资料，称此矩阵为稀疏矩阵（sparse matrix），由于矩阵在程式中常使用二维阵列表示，二维阵列的大小与使用的记忆体空间成正比，如果多数的元素没有资料，则会造成记忆体空间的浪费，为 此，必须设计稀疏矩

阵的阵列储存方式，利用较少的记忆体空间储存完整的矩阵资讯。

解法

在这边所介绍的方法较为简单，阵列只储存矩阵的行数、列数与有资料的索引位置及其值，在需要使用矩阵资料时，再透过程式运算加以还原，例如若矩阵资料如下 ，其中0表示矩阵中该位置没有资料：

```
0 0 0 0 0 0
0 3 0 0 0 0
0 0 0 6 0 0
0 0 9 0 0 0
0 0 0 0 12 0
```

这个矩阵是5X6矩阵，非零元素有4个，您要使用的阵列第一列记录其列数、行数与非零元素个数：

```
5 6 4
```

阵列的第二列起，记录其位置的列索引、行索引与储存值：

```
1 1 3
2 3 6
3 2 9
4 4 12
```

所以原本要用30个元素储存的矩阵资讯，现在只使用了15个元素来储存，节省了不少记忆体的使用。

C

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

int main(void) {
    int num[5][3] = {{5, 6, 4},
                     {1, 1, 3},
                     {2, 3, 6},
                     {3, 2, 9},
                     {4, 4, 12}};
    int i, j, k = 1;

    printf("sparse matrix：\n");
    for(i = 0; i < 5; i++) {
        for(j = 0; j < 3; j++) {
            printf("%4d", num[i][j]);
        }
        putchar('\n');
    }

    printf("\nmatrix还原：\n");
    for(i = 0; i < num[0][0]; i++) {
        for(j = 0; j < num[0][1]; j++) {
            if(k < num[0][2] &&
               i == num[k][0] && j == num[k][1]) {
                printf("%4d ", num[k][2]);
                k++;
            }
            else
```

```
                printf("%4d ", 0);
            }
            putchar('\n');
        }

        return 0;
}
```

47.多维矩阵转一维矩阵

说明

有的时候，为了运算方便或资料储存的空间问题，使用一维阵列会比二维或多维阵列来得方便，例如上三角矩阵、下三角矩阵或对角矩阵，使用一维阵列会比使用二维阵列来得节省空间。

解法

以二维阵列转一维阵列为例，索引值由0开始，在由二维阵列转一维阵列时，我们有两种方式：「以列（Row）为主」或「以行（Column）为主」。由于 C/C++、Java等的记忆体配置方式都是以列为主，所以您可能会比较熟悉前者（Fortran的记忆体配置方式是以行为主）。

以列为主的二维阵列要转为一维阵列时，是将二维阵列由上往下一列一列读入一维阵列，此时索引的对应公式如下所示，其中row与column是二维阵列索引，loc表示对应的一维阵列索引：

```
loc = column + row*行数
```

以行为主的二维阵列要转为一维阵列时，是将二维阵列由左往右一行一行读入一维阵列，此时索引的对应公式如下所示：

```
loc = row + column*列数
```

公式的推导您画图看看就知道了，如果是三维阵列，则公式如下所示，其中i（个数u1）、j（个数u2）、k（个数u3）分别表示三维阵列的三个索引：

```
以列为主：loc = i*u2*u3 + j*u3 + k
以行为主：loc = k*u1*u2 + j*u1 + i
```

更高维度的可以自行依此类推，但通常更高维度的建议使用其它资料结构（例如物件包装）会比较具体，也不易搞错。

在C/C++中若使用到指标时，会遇到指标运算与记忆体空间位址的处理问题，此时也是用到这边的公式，不过必须在每一个项上乘上资料型态的记忆体大小。

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

int main(void) {
    int arr1[3][4] = {{1, 2, 3, 4},
                      {5, 6, 7, 8},
                      {9, 10, 11, 12}};
    int arr2[12] = {0};
    int row, column, i;
```

```
    printf("原二维资料：\n");
    for(row = 0; row < 3; row++) {
        for(column = 0; column < 4; column++) {
            printf("%4d", arr1[row][column]);
        }
        printf("\n");
    }

    printf("\n以列为主：");
    for(row = 0; row < 3; row++) {
        for(column = 0; column < 4; column++) {
            i = column + row * 4;
            arr2[i] = arr1[row][column];
        }
    }
    for(i = 0; i < 12; i++)
        printf("%d ", arr2[i]);

    printf("\n以行为主：");
    for(row = 0; row < 3; row++) {
        for(column = 0; column < 4; column++) {
            i = row + column * 3;
            arr2[i] = arr1[row][column];
        }
    }
    for(i = 0; i < 12; i++)
        printf("%d ", arr2[i]);

    printf("\n");

    return 0;
}
```

48.上三角、下三角、对称矩阵

说明

上三角矩阵是矩阵在对角线以下的元素均为0，即$A_{ij} = 0$，$i > j$，例如：

```
1  2  3   4   5
0  6  7   8   9
0  0  10   11  12
0  0  0   13  14
0  0  0   0  15
```

下三角矩阵是矩阵在对角线以上的元素均为0，即$A_{ij} = 0$，$i < j$，例如：

```
 1  0  0  0  0
 2  6  0  0  0
 3  7  10 0  0
```

```
 4  8  11 13 0
 5  9  12 14 15
```

对称矩阵是矩阵元素对称于对角线，例如：

```
 1  2  3  4  5
 2  6  7  8  9
 3  7  10 11 12
 4  8  11 13 14
 5  9  12 14 15
```

上三角或下三角矩阵也有大部份的元素不储存值（为0），我们可以将它们使用一维阵列来储存以节省储存空间，而对称矩阵因为对称于对角线，所以可以视为上三角或下三角矩阵来储存。

解法

假设矩阵为nxn，为了计算方便，我们让阵列索引由1开始，上三角矩阵化为一维阵列，若以列为主，其公式为：loc = n*(i-1) - i*(i-1)/2 + j
化为以行为主，其公式为：loc = j*(j-1)/2 + i

下三角矩阵化为一维阵列，若以列为主，其公式为：loc = i*(i-1)/2 + j

若以行为主，其公式为：loc = n*(j-1) - j*(j-1)/2 + i
公式的导证其实是由等差级数公式得到，您可以自行绘图并看看就可以导证出来，对于C/C++或Java等索引由0开始的语言来说，只要将i与j各加1，求得loc之后减1即可套用以上的公式。

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

int main(void) {
    int arr1[N][N] = {
        {1, 2, 3,  4,    5},
        {0, 6, 7,  8,    9},
        {0, 0, 10, 11, 12},
        {0, 0, 0,   13, 14},
        {0, 0, 0,   0,   15}};

    int arr2[N*(1+N)/2] = {0};

    int i, j, loc = 0;

    printf("原二维资料：\n");
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) {
            printf("%4d", arr1[i][j]);
        }
        printf("\n");
    }

    printf("\n以列为主：");
    for(i = 0; i < N; i++) {
```

```
            for(j = 0; j < N; j++) {
                if(arr1[i][j] != 0)
                    arr2[loc++] = arr1[i][j];
            }
        }
        for(i = 0; i < N*(1+N)/2; i++)
            printf("%d ", arr2[i]);

        printf("\n输入索引(i, j)：");
        scanf("%d, %d", &i, &j);
        loc = N*i - i*(i+1)/2 + j;
        printf("(%d, %d) = %d", i, j, arr2[loc]);

        printf("\n");
        return 0;
}
```

49.奇数魔方阵

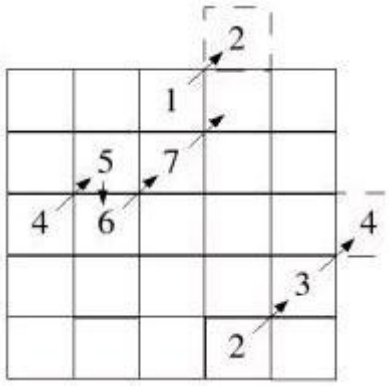
说明

将1到n(为奇数)的数字排列在nxn的方阵上，且各行、各列与各对角线的和必须相同，如下所示：

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

解法

填魔术方阵的方法以奇数最为简单，第一个数字放在第一行第一列的正中央，然后向右(左)上填，如果右(左)上已有数字，则向下填，如下图所示：



一般程式语言的阵列索引多由0开始，为了计算方便，我们利用索引1到n的部份，而在计算是向右(左)上或向下时，我们可以将索引值除以n值，如果得到余数为1就向下，否则就往右(左)上，原理很简单，看看是不是已经在同一列上绕一圈就对了。

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

```
#define N 5

int main(void) {
	int i, j, key;
	int square[N+1][N+1] = {0};

	i = 0;
	j = (N+1) / 2;

	for(key = 1; key <= N*N; key++) {
		if((key % N) == 1)
			i++;
		else {
			i--;
			j++;
		}

		if(i == 0)
			i = N;
		if(j > N)
			j = 1;

		square[i][j] = key;
	}

	for(i = 1; i <= N; i++) {
		for(j = 1; j <= N; j++)
			printf("%2d ", square[i][j]);
	}

	return 0;
}
```

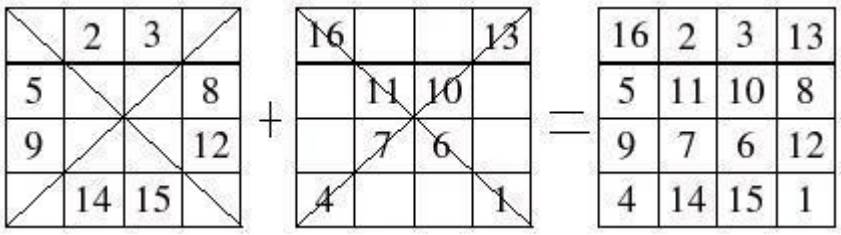
50.4N 魔方阵

说明

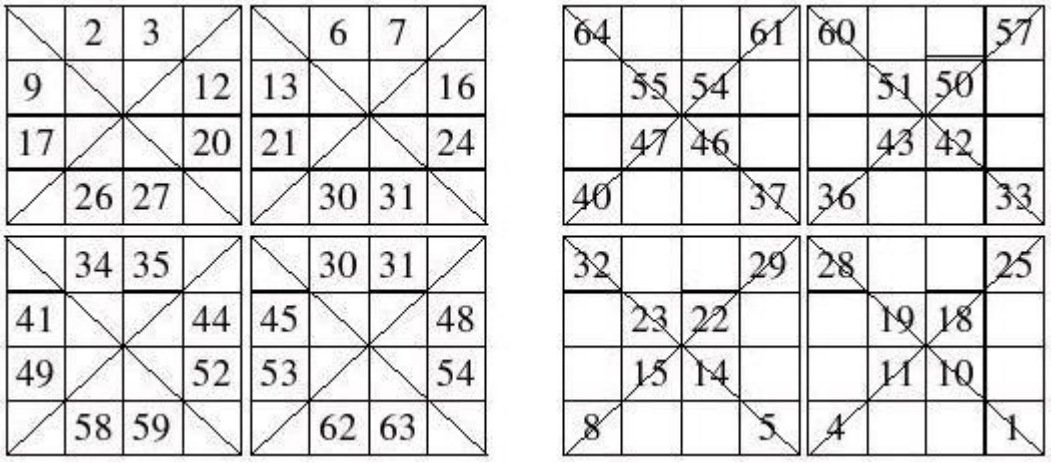
与 奇数魔术方阵 相同，在于求各行、各列与各对角线的和相等，而这次方阵的维度是4的倍数。

解法

先来看看4X4方阵的解法：



简单的说，就是一个从左上由1依序开始填，但遇对角线不填，另一个由左上由16开始填，但只填在对角线，再将两个合起来就是解答了；如果N大于2，则以 4X4为单位画对角线：



至于对角线的位置该如何判断，有两个公式，有兴趣的可以画图印证看看，如下所示：

左上至右下：j % 4 == i % 4

右上至左下：(j % 4 + i % 4) == 1

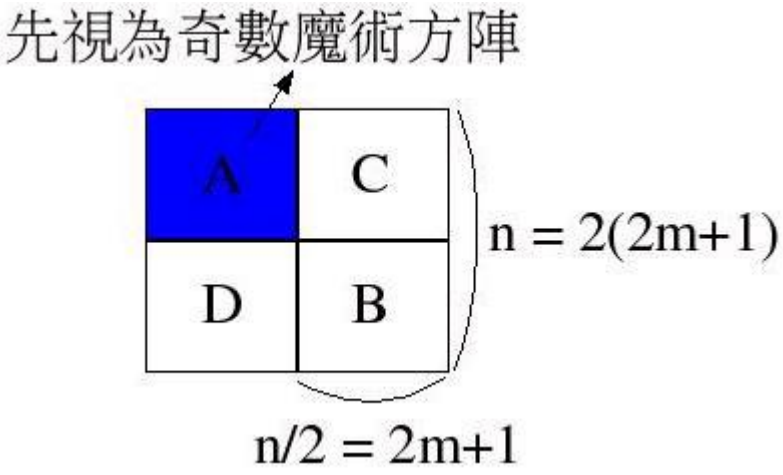
```
#include <stdio.h>
#include <stdlib.h>

#define N 8

int main(void) {
    int i, j;
    int square[N+1][N+1] = {0};

    for(j = 1; j <= N; j++) {
        for(i = 1; i <= N; i++){
            if(j % 4 == i % 4 || (j % 4 + i % 4) == 1)
                square[i][j] = (N+1-i) * N -j + 1;
            else
                square[i][j] = (i - 1) * N + j;
        }
    }

    for(i = 1; i <= N; i++) {
        for(j = 1; j <= N; j++)
            printf("%2d ", square[i][j]);
        printf("\n");
    }
```

```
    return 0;
}
```

51.2(2N+1) 魔方阵

说明方阵的维度整体来看是偶数，但是其实是一个奇数乘以一个偶数，例如6X6，其中6=2X3，我们也称这种方阵与单偶数方阵。

解法如果您会解奇数魔术方阵，要解这种方阵也就不难理解，首先我们令n=2(2m+1)，并将整个方阵看作是数个奇数方阵的组合，如下所示：



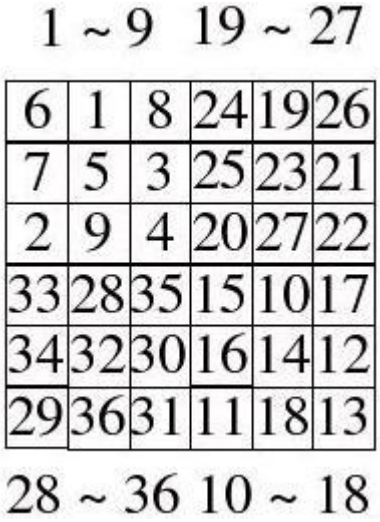
首先依序将A、B、C、D四个位置，依奇数方阵的规则填入数字，填完之后，方阵中各行的和就相同了，但列与对角线则否，此时必须在A-D与C- B之间，作一些对应的调换，规则如下：

将A中每一列(中间列除外)的头m个元素，与D中对应位置的元素调换。

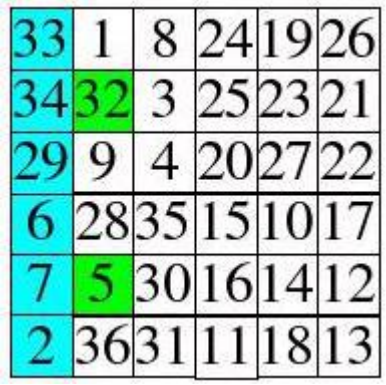
将A的中央列、中央那一格向左取m格，并与D中对应位置对调

将C中每一列的倒数m-1个元素，与B中对应的元素对调

举个实例来说，如何填6X6方阵，我们首先将之分解为奇数方阵，并填入数字，如下所示：



接下来进行互换的动作，互换的元素以不同颜色标示，如下：



33	1	8	24	19	26
34	32	3	25	23	21
29	9	4	20	27	22
6	28	35	15	10	17
7	5	30	16	14	12
2	36	31	11	18	13

由于m-1的数为0，所以在这个例子中，C-B部份并不用进行对调。

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

#define N 6
#define SWAP(x,y) {int t; t = x; x = y; y = t;}

void magic_o(int [][N], int);
void exchange(int [][N], int);

int main(void) {
    int square[N][N] = {0};
    int i, j;

    magic_o(square, N/2);
    exchange(square, N);

    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++)
            printf("%2d ", square[i][j]);
        printf("\n");
    }

    return 0;
}

void magic_o(int square[][N], int n) {
    int count, row, column;

    row = 0;
    column = n / 2;

    for(count = 1; count <= n*n; count++) {
        square[row][column] = count;                    // 填A
        square[row+n][column+n] = count + n*n;    // 填B
        square[row][column+n] = count + 2*n*n;    // 填C
        square[row+n][column] = count + 3*n*n;    // 填D
        if(count % n == 0)
            row++;
```

```
        else {
            row = (row == 0) ? n - 1 : row - 1 ;
            column = (column == n-1) ? 0 : column + 1;
        }
    }
}

void exchange(int x[][N], int n) {
    int i, j;
    int m = n / 4;
    int m1 = m - 1;

    for(i = 0; i < n/2; i++) {
        if(i != m)   {
            for(j = 0; j < m; j++)                  // 处理规则 1
                SWAP(x[i][j], x[n/2+i][j]);
            for(j = 0; j < m1; j++)                 // 处理规则 2
                SWAP(x[i][n-1-j], x[n/2+i][n-1-j]);
        }
        else {   // 处理规则 3
            for(j = 1; j <= m; j++)
                SWAP(x[m][j], x[n/2+m][j]);
            for(j = 0; j < m1; j++)
                SWAP(x[m][n-1-j], x[n/2+m][n-1-j]);
        }
    }
}
```