学号：19S103256

姓名：文荟俨

实验一：分治算法

1. 实验目的

(1)掌握分治算法的设计思想与方法
(2)熟练使用高级编程语言实现分治算法
(3)通过对比简单算法以及不同的分治求解思想，理解算法复杂度

2. 实验问题

求解凸包问题。输入:平面上 n 个点的集合 Q；输出:Q 的凸包。其中，Q 的凸包是一个凸多边形 P，Q 的点在 P 上或 P 内。凸多边形指连接它任意两点的边都在其内部。

3. 实验步骤

(1) 蛮力法
算法 **Bruteforce**(Q)
输入 :平面上 *n* 个点的集合 Q
输出 :CH(Q)，Q 的凸包

1: For ∀A,B,C,D∈Q **Do**
2: If D 位于 ABC 组成的三角形内(根据面积判定)
3: **Then** 从 Q 中删除该点
4: A←Q 中横坐标最大的点
5: B←Q 中横坐标最小的点
6: S_L←{P|P∈Q 且 P 位于 AB 直线的下方}
7: S_U←{P|P∈Q 且 P 位于 AB 直线的上方}
8: 对 S_L 和 S_U 分别升序排序
9: 输出 A, S_L, B, 逆序 S_U

算法主要时间消耗在第 1-3 行，对 ABCD 点进行循环，复杂度为 $O(n^4)$。判定 D 点是否在 ABC 三角形内部，采用面积法，计算时间为常数，具体如图 1 所示。若 $S_{ABD}+S_{BCD}+S_{ADC}=S_{ABC}$,则 D 点位于△ABC 内部，应该删除。

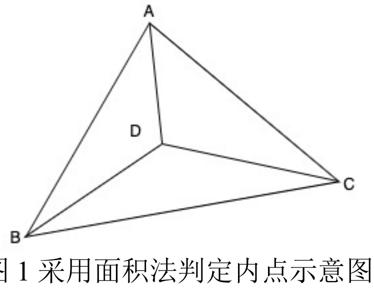

图 1 采用面积法判定内点示意图

这里可利用叉积计算三角形面积，核心代码如清单 1 所示。

清单 1 利用叉积计算三角形面积

```
float GetTriangleSquar(const point_float pt0, const point_float pt1,
const point_float pt2)
{
	point_float AB, BC;
	AB.x = pt1.x - pt0.x;
	AB.y = pt1.y - pt0.y;
	BC.x = pt2.x - pt1.x;
	BC.y = pt2.y - pt1.y;
	return fabs((AB.x * BC.y - AB.y * BC.x)) / 2.0f;
}

bool IsInTriangle(const point_float A, const point_float B, const
point_float C, const point_float D)
{
     if (A.x == -1 || B.x == -1 || C.x == -1 || D.x == -1) return false;
	float SABC, SADB, SBDC, SADC;
	SABC = GetTriangleSquar(A, B, C);
	SADB = GetTriangleSquar(A, D, B);
	SBDC = GetTriangleSquar(B, D, C);
	SADC = GetTriangleSquar(A, D, C);

	float SumSuqar = SADB + SBDC + SADC;

	if ((-ABS_FLOAT_0 < (SABC - SumSuqar)) && ((SABC - SumSuqar) <
ABS_FLOAT_0))
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

(2) Graham-Scan

算法 **Graham-Scan**(Q)

输入 :平面上 *n* 个点的集合 Q

输出 :CH(Q)，Q 的凸包

1:	求 Q 中 y 坐标最小的点 p_0，若存在多个点，则选 x 坐标最小的
2:	按照与 p_0 的极角逆时针排序其余点< p_0, p_1, …, p_m>，
3:	如果存在极角相同的点，选择欧式距离更近的点在前
4:	S.push(p_0), S.push(p_1), S.push(p_2)
5:	**For** i←3 To m **Do**
6:	**While** S.next_to_top(), S.Top()和 p_i 形成非左 **Do**
7:	S.Pop()
8:	S.push(p_i)
9:	**Return** S

算法第 1 步需要线性时间，第 2 步排序需要 O(nlogn)，第 4-7 步需要线性时间，总的时间复杂度为 O(nlogn)。判定是否非左可采用叉积，如图 2 所示。给定 **A**=(a, b), **B**=(c, d)，若 ac-bd>0，则 **B** 在 **A** 的左侧，反之在右侧，等于 0 时共线。

图 2 叉积判定向量非左示意图

算法扫描过程的核心代码如清单 2 所示。

清单 2 GrahamScan 核心代码

```
min_node.y = 100;
for (int i = 0; i < dotnum; i++) {
	if (dot[i * 2 + 1] <= min_node.y) {
		if (dot[i * 2 + 1] == min_node.y) {
			if (dot[i * 2] < min_node.x) {
				min_node.x = dot[i * 2];
				min_node.y = dot[i * 2 + 1];
			}
		}
		else {
			min_node.x = dot[i * 2];
			min_node.y = dot[i * 2 + 1];
		}
	}
}
struct node *dot_angle = (struct node *)malloc(dotnum * sizeof(struct
node));

for (int i = 0; i < dotnum; i++) {
	if (dot[i * 2] != min_node.x || dot[i * 2 + 1] != min_node.y) {
		   //dot_angle[i].r = atan2(dot[i * 2 + 1], -dot[i * 2]);
			dot_angle[i].x = dot[i * 2];
		dot_angle[i].y = dot[i * 2 + 1];
	}
	else {
		//dot_angle[i].r = -max_num;
		dot_angle[i].x = dot[i * 2];
		dot_angle[i].y = dot[i * 2 + 1];
	}
}
sort(dot_angle, dot_angle+dotnum, compare);
stack<int> s;
s.push(0);
s.push(1);
s.push(2);
int next_to_top, top;
for (int i = 3; i < dotnum; i++) {
	Print_process(dotnum , i, 1);
```

```
        while (cross(dot_angle[next_top(s)], dot_angle[s.top()], 
dot_angle[i])>=0){
                s.pop();
                if (s.size() == 1) break;
        }
        s.push(i);
}
end = clock();
int *convex = (int *)malloc((s.size()*2) * sizeof(int));
int convex_num = 0;
while (!s.empty()) {
        convex[convex_num * 2] = dot_angle[s.top()].x;
        convex[convex_num * 2 + 1] = dot_angle[s.top()].y;
        convex_num++;
        s.pop();
}
```

(3)分治法

算法 **Divide_and_conquer**(Q)

输入 :平面上 n 个点的集合 Q

输出 :CH(Q)，Q 的凸包

- Preprocess:找到横坐标最大的点 A 和最小的点 B，标记每个点是否访问，若全部访问则算法停止。(时间复杂度为 $O(n)$);
- Divide:作直线 AB，把凸包分为 S_L 和 S_U 上下两个子集，对每个部分求得点 P_{max},使得 S_{ABPmax} 最大。将三角形三个点和三角形内部的点标记为已访问，删去所有三角形内部及边上的点。(时间复杂度为 $O(n)$);
- Conquer: 进一步依据△ABP_{max} 划分成左右两个部分，当作 S_L 和 S_U，分治递归、不断重复。(时间复杂度为 $2T(n/2)$)。

由 master 定理解得算法总时间复杂度为 O(nlogn)。算法的原理示意图如图 3，核心代码如清单 3。

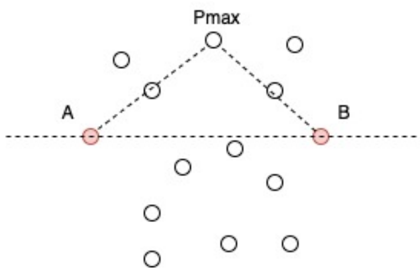

图 3 分治法原理示意图

清单 3 分治法核心代码

```
void Divide(int first, int last, vector<Point> &node, int *vst)
{
        int max_p = 0, index = -1;
        int i = first;
        if (first < last)
        {
                for (i = first + 1; i < last; i++)
                {
                        int calcu = Cal_S(node[first], node[i], node[last]);
                        if (calcu > max_p)
```

```
                {
                        max_p = calcu;
                        index = i;
                }

        }
    }
    else
    {
        for (i - 1; i >last; i--)
        {
            int calcu = Cal_S(node[first], node[i], node[last]);
            if (calcu > max_p)
            {
                max_p = calcu;
                index = i;
            }
        }
    }
    if (index != -1)
    {
        vst[index] = 1;
        Divide(first, index, node, vst);
        Divide(index, last, node, vst);
    }
}


void divide_and_conquer(char* init_path) {
    const char *img_path;
    const char *dot_path;
    int dotnum, whether_show_img;
    map<string, string>fname;
    CParseIniFile config;
    bool flag = config.ReadConfig(init_path, fname, "4");
    if (flag) {
        dotnum = stoi(fname["dotnum"]);
        whether_show_img = stoi(fname["whether_show_img"]);
        img_path = fname["img_path"].c_str();
        dot_path = fname["dot_path"].c_str();
    }
    else {
        cout << "Loading ini 4 error!" << endl;
        return;
    }
    //load dots
    int *dot = (int *)malloc((dotnum * 2) * sizeof(int));
    if (dotnum < 3) {
        cout << "Please generate more than 3 dots." << endl;
        exit(-1);
    }
```

```
    Load_file(dot_path, dot);
    int *vst = (int *)malloc(dotnum * sizeof(int));

    clock_t begin, end;
    begin = clock();
    vector<Point> node;
    Point temp_node;
    for (int i = 0; i < dotnum; i++) {
        vst[i] = 0;
        temp_node.x = dot[2 * i];
        temp_node.y = dot[2 * i + 1];
        node.push_back(temp_node);
    }
    vst[0] = 1;
    vst[dotnum - 1] = 1;
    sort(node.begin(), node.end(), cmp_x);
    Divide(0, dotnum - 1, node, vst);
    Divide(dotnum - 1, 0, node, vst);
    int convex_num = 0;
    for (int i = 0; i < dotnum; i++) {
        if (vst[i] == 1) convex_num++;
    }
    int *convex = (int *)malloc((convex_num * 2) * sizeof(int));
    //cout << "convex_num:" << convex_num << endl;
    convex_num = 0;
    for (int i = 0; i < dotnum; i++) {
        if (vst[i] == 1) {
            convex[2 * convex_num] = node[i].x;
            convex[2 * convex_num + 1] = node[i].y;
            convex_num++;
        }
    }
    end = clock();
    cout << "convex_num:" << convex_num << endl;
    plot(dot, convex, img_path, dotnum, convex_num, whether_show_img);
    free(convex);
    free(dot);
    cout << "Calculation time:" << end - begin << endl;
}
```

(4)绘图算法

算法 **DrawResult**(A,B, S_L,S_U)

输入 : 凸包算法求得的 A,B, S_L 和 S_U

输出 : 点阵图，通过连线将凸包连接起来，形成闭环

1:	对 S_L 和 S_U 分别升序排序
2:	可视化作图，连接 A 点和 S_L 中横坐标最小的点
3:	逆时针连接 S_L 中的点，顺序
4:	连接 B 点和 S_L 中横坐标最大的点
5:	连接 B 点和 S_U 中横坐标最大的点

6:	逆时针连接 S_L 中的点，逆序
7:	连接 A 点和 S_U 中横坐标最小的点

算法的核心代码如清单 4 所示。

清单 4 绘图算法核心代码

```
Mat img = Mat::zeros(Size(100* large_size, 100* large_size), CV_8UC3);
for (int m = 0; m < 100; m++) {
	for (int n = 0; n < 100; n++) {
		drawBlock(img, m, n, large_size, large_size, wall_color);
	}
}

for (int i = 0; i < dotnum; i++) {
	drawBlock(img, dot[i * 2 + 1], dot[i * 2], large_size, large_size, 
Scalar(0, 0, 255));
}

if (convex_num > 3) {
	//连线
	if (down_node_count) {
		line(img, Point(min_node[0] * large_size + gap_temp, 
min_node[1] * large_size + gap_temp), Point(down_node[0][0] * large_size 
+ gap_temp, down_node[0][1] * large_size + gap_temp), Scalar(0, 255, 0), 
line_size);
		cout << min_node[0] << " " << min_node[1] << "," << 
down_node[0][0] << " " << down_node[0][1] << endl;
		if (down_node.size() > 1) {
			for (int i = 0; i < down_node.size() - 1; i++) {
				line(img, Point(down_node[i][0] * large_size + 
gap_temp, down_node[i][1] * large_size + gap_temp), Point(down_node[i + 
1][0] * large_size + gap_temp, down_node[i + 1][1] * large_size + 
gap_temp), Scalar(0, 255, 0), line_size);
			}
		}
		line(img, Point(down_node[down_node.size() - 1][0] * 
large_size + gap_temp, down_node[down_node.size() - 1][1] * large_size + 
gap_temp), Point(max_node[0] * large_size + gap_temp, max_node[1] * 
large_size + gap_temp), Scalar(0, 255, 0), line_size);
	}
	else
		line(img, Point(min_node[0] * large_size + gap_temp, 
min_node[1] * large_size + gap_temp), Point(max_node[0] * large_size + 
gap_temp, max_node[1] * large_size + gap_temp), Scalar(0, 255, 0), 
line_size);
	if (up_node_count) {
		line(img, Point(up_node[up_node.size() - 1][0] * large_size 
+ gap_temp, up_node[up_node.size() - 1][1] * large_size + gap_temp), 
Point(max_node[0] * large_size + gap_temp, max_node[1] * large_size + 
gap_temp), Scalar(0, 255, 0), line_size);
		if (up_node_count > 1) {
			for (int i = 0; i < up_node.size() - 1; i++) {
				line(img, Point(up_node[i][0] * large_size + 
gap_temp, up_node[i][1] * large_size + gap_temp), Point(up_node[i + 1][0] 
* large_size + gap_temp, up_node[i + 1][1] * large_size + gap_temp), 
Scalar(0, 255, 0), line_size);
			}
```

```
            }
            line(img, Point(min_node[0] * large_size + gap_temp,
min_node[1] * large_size + gap_temp), Point(up_node[0][0] * large_size +
gap_temp, up_node[0][1] * large_size + gap_temp), Scalar(0, 255, 0),
line_size);
      }


}
else {
      line(img, Point(convex[0 * 2] * large_size + gap_temp, convex[0 *
2 + 1] * large_size + gap_temp), Point(convex[1 * 2] * large_size +
gap_temp, convex[1 * 2 + 1] * large_size + gap_temp), Scalar(0, 255, 0),
line_size);
      line(img, Point(convex[1 * 2] * large_size + gap_temp, convex[1 *
2 + 1] * large_size + gap_temp), Point(convex[2 * 2] * large_size +
gap_temp, convex[2 * 2 + 1] * large_size + gap_temp), Scalar(0, 255, 0),
line_size);
      line(img, Point(convex[2 * 2] * large_size + gap_temp, convex[2 *
2 + 1] * large_size + gap_temp), Point(convex[0 * 2] * large_size +
gap_temp, convex[0 * 2 + 1] * large_size + gap_temp), Scalar(0, 255, 0),
line_size);
}
imwrite(path, img);
```

4. 实验结果与分析

(1)实验参数配置

本实验将 exe 程序和 config 配置文件抽离开来，能够实现较好的交互，对配置文件的说明如清单 5 所示。我们可以在 method 处选择功能，分别有生成数据集、暴力求解、Graham Scan、分治法和 Opencv 库 4 个功能，具体的参数可以在相应功能区下面配置。

清单 5 config.ini 文件

```
[method]
; 0: generate node
; 1: bruteforce
; 2: Graham scan
; 3: opencv
; 4: divide-and-conquer
method = 3

[0]
;生成随机点
dotnum=6000
dot_path = L:/why_workspace/cpl/txt/dot6000.txt

[1]
;读入随机点
dotnum=300
dot_path = L:/why_workspace/cpl/txt/dot300.txt

;是否可视化凸包
whether_show_img = 1
```

```
;仅在 whether_show_img = 1 时有效
img_path = L:/why_workspace/cpl/txt/test.jpg

[2]
;读入随机点
dotnum=4000
dot_path = L:/why_workspace/cpl/txt/dot4000.txt
;是否可视化凸包
whether_show_img = 1
;仅在 whether_show_img = 1 时有效
img_path = L:/why_workspace/cpl/txt/test2.jpg

[3]
;读入随机点
dotnum=100
dot_path = L:/why_workspace/cpl/txt/dot100.txt
;是否可视化凸包
whether_show_img = 1
;仅在 whether_show_img = 1 时有效
img_path = L:/why_workspace/cpl/txt/test3.jpg

[4]
;读入随机点
dotnum=4000
dot_path = L:/why_workspace/cpl/txt/dot4000.txt
;是否可视化凸包
whether_show_img = 1
;仅在 whether_show_img = 1 时有效
img_path = L:/why_workspace/cpl/txt/test4.jpg
```

(2)结果可视化展示

当节点个数取 300 时，用暴力求解法、grahamscan、分治法和 opencv convexHull 库分别得到如图 4 的 10 个凸包，四者结果相同，算法实现正确。进一步将其可视化展示验证，得到图 5 所示结果，算法正确。

```
The progeress:0.99/1.00
convex_num:10
(0,79) (4,98) (34,99) (68,98) (85,97) (99,85) (99,1) (68,0) (3,0) (0,34)
Saved img result successfully.
Calculation time:5721
```

图 4-a 暴力求解结果(计算时间以 ms 为单位，下同)

```
The progeress:1.00/1.00
convex_num:10
(0,79) (4,98) (34,99) (68,98) (85,97) (99,85) (99,1) (68,0) (3,0) (0,34)
Saved img result successfully.
Calculation time:12
```

图 4-b GrahamScan 结果

```
convex_num:10
(0,34) (0,79) (4,98) (34,99) (68,98) (85,97) (99,85) (99,1) (68,0) (3,0)
Saved img result successfully.
Calculation time:0
```

图 4-c 分治法结果

```
convex_num:10
(0,34) (0,79) (4,98) (34,99) (68,98) (85,97) (99,85) (99,1) (68,0) (3,0)
Saved img result successfully.
Calculation time:0
```

图 4-d Opencv 验证结果

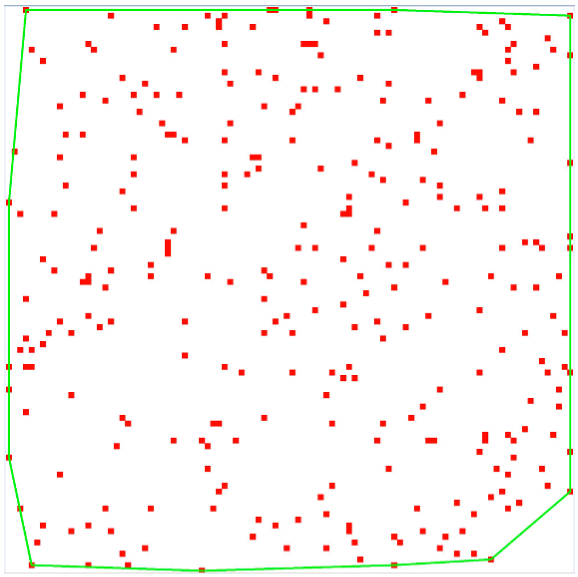

图 5 上述结果可视化展示

(3)性能对比

为保证结果准确性,我们进行了 10 次数据集生成,分别进行了实验,取平均值,得到了图 6 所示结果。由于暴力求解法速度过慢，该方法仅展示了数据量从 100-1000 时的实验结果。

(4)总结

在数据量较小时，分治法、Graham Scan 和暴力求解法差距不明显，但当数据量上去过后，特别是大于 1000 个点时，分治法具有显著的优势,Graham Scan 速度有所增加，但可以接受,而暴力求解法的时间则随数据量呈指数增长。经过验证，本实验实现的分治法具有和 Opencv convecHull 库几乎一样的速度，可以推测官方的库也是采用的类似的分治法。

5. 实验心得

这次实验遇到的坑挺多的，首先是判断三角形内点，最开始我是参考的老师给的判断直线方向，计算 g 值相乘>0 来判定是否处于三角形内部，结果总是有小问题，主要是因为 Opencv 的坐标轴是 y 轴向下的，导致很难想明白。后来采用了面积法，方便了很多。其次就是计算 graham scan 算法的非左方向，我自己写了一个算法，结果总是考虑不周全一些特殊情况，后来采用了叉积计算，

简便了很多。再就是对极角排序时，没有考虑到极角相同的点，对于这种情况，应该根据欧氏距离排序，把离得近的放在前。最后是对初始点的选取，通常是选取 y 轴最小的点作为起始点，但是假设存在多个最小的 y 值时，如果选取到的 y 值位于中间，不是这些点中 x 值最小或最大的，那么最后的凸包就会存在冗余，多了起始点。而事实上，起始点是处于两个点之间的，可以被另外 2 个极点线性表达。

总的来说，实验一写的很痛苦，花了不少时间，但是写完过后还是挺有成就感的，特别是发现自己实现的分治法具有和 Opencv 的库几乎一样的效率。

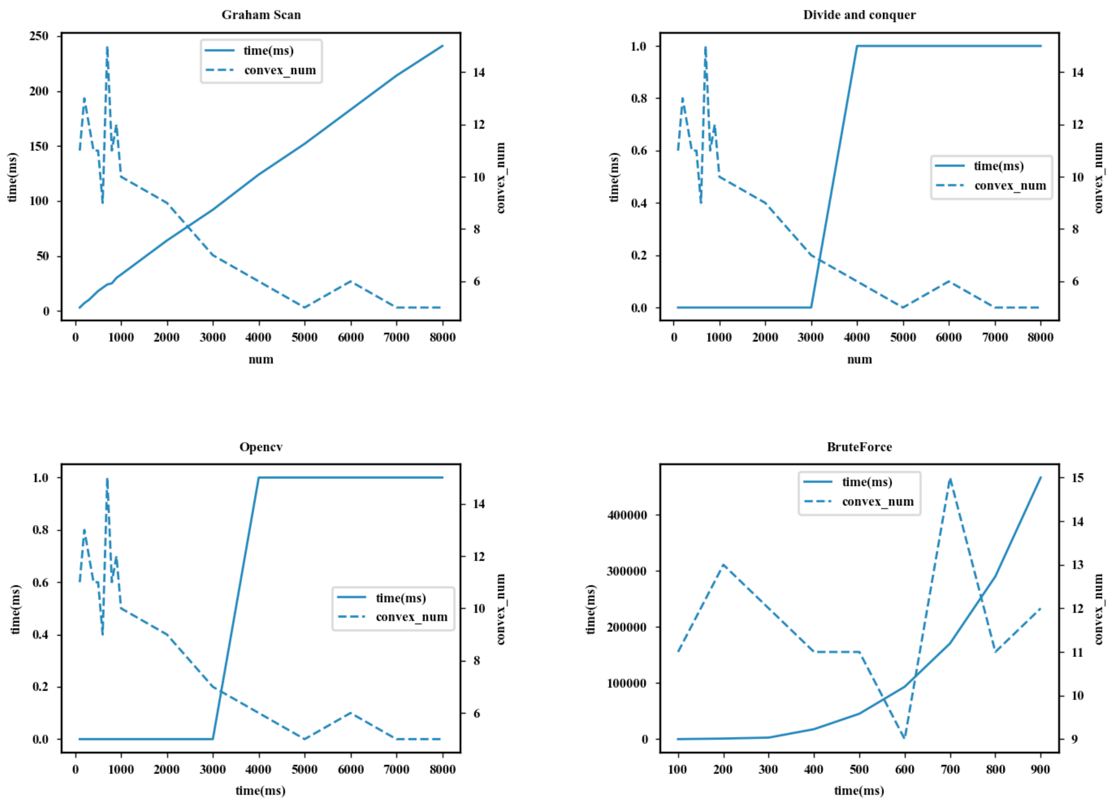

图 6 各方法性能对比