目 录

1 选择排序 **2**

2 冒泡排序 **2**

3 顺序查找 **3**

4 蛮力字符串匹配 **3**

5 最近对问题 **4**

6 凸包问题 **5**

7 穷举查找 **7**

7.1 旅行商问题 7

7.2 背包问题 7

7.3 分配问题 7

1 选择排序

算法过程：对列表做第 i 遍扫描的时候，该算法在最后 n-i 个元素中寻找最小元素，然后拿它和 A_i 交换。

代码实现如下：

```
void select_sort(int *A, int len)
{
    int min;
    int i;
    int j;
    int temp;
    for(i = 0; i < len-1; ++i)
    {
        min = i;
        for(j = i + 1; j < len; ++j)
        {
            if(A[j] < A[min])
                min = j;
        }
        temp = A[min];
        A[min] = A[i];
        A[i] = temp;
    }
}
```

算法分析：

- 该算法的基本操作是键值比较 A[i]<A[min]。该比较执行次数为 $C(n) = \sum_{i=0} n - 2(n-1-i)$。
- 键的交换次数仅为 $\Theta(n)$。
- 选择排序最坏情况和平均情况的复杂度为 $\Theta(n^2)$。

2 冒泡排序

算法思路：从第一个元素开始，比较列表中相邻元素，如果它们是逆序的就交换它们的位置。重复这个过程 n-1 遍，这个列表就排好序了。

实现代码如下：

```
void bubble_sort(int *A, int len)
{
    int i, j;
    int temp;
    for(i = 0; i < len; ++i)
    {
        for(j = 0; j < len-i-1; ++j)
```

```
                {
                    if(A[j+1] < A[j])
                    {
                        temp = A[j];
                        A[j] = A[j+1];
                        A[j+1] = temp;
                    }
                }
            }
        }
```

算法分析：

- 冒泡排序的键值比较次数为 $\frac{(n-1)n}{2} \in \Theta(n^2)$。
- 冒泡排序的最坏情况和平均情况的复杂度为 $\Theta(n^2)$。

3 顺序查找

查找问题就是在给定的集合中找一个给定的值。

算法思路：将给定序列中的连续元素和给定的查找健作比较，直到遇到一个匹配的元素。

算法实现的小技巧：将查找键添加到列表的末尾，那么查找一定会成功，所以就不必在算法的每次循环时都检查是否到达了表的末尾。

代码实现：

```
int sequential_search(int *A, int len, int key)
{
    A[len] = key;
    int i = 0;
    while(A[i] != key)
        ++i;
    if(i < n)
        return i;
    else
        return -1;
}
```

算法分析：顺序查找是线性算法，时间复杂度为 $\Theta(n)$。

4 蛮力字符串匹配

字符串匹配问题：给定一个 n 个字符组成的串，称为文本。给定一个 m 个字符的串，称为模式。

算法思路：将模式对准文本的前 m 个字符，然后从左到右匹配每一对相应的字符，直到 m 个字符全部匹配。如果遇到不匹配的字符，将模式向右移一位，然后从模式的第一个字符开始匹配。

算法实现：

```
void brute_force_string_match(int *Text, int *pattern, int n, int m)
{
    int i,j;
    for(i = 0; i < n-m; ++i)
    {
        j = 0;
        while((j < m) && (P[j] == Text[i+j]))
            ++j;
        if(j == m)
            return i;
    }
    return -1;
}
```

算法分析：

- 最坏的情况下，算法复杂度为 $\Theta(nm)$。
- 算法的平均时间复杂度为 $\Theta(n+m) = \Theta(n)$。

5 最近对问题

最近对问题：找出一个包含 n 个点的集合中距离最近的两个点。

算法思路：分别计算每一对点之间的距离，然后找出距离最小的那一对。为了不对同一对点计算两次距离，我们只考虑 i<j 的那些对 $(P_i P_j)$。

算法实现：

```
struct Point
{
    int x;
    int y;
};
typedef struct Point Point;
int* brute_force_closest_points(Point *P, int n)
{
    int x1, x2;
    int y1, y2;
    int i, j;
    int d;
    int dmin = INT32_MAX;
    int index[2];
    for(i = 0; i < n; ++i)
    {
        x1 = P[i].x;
```

```
            y1 = P[i].y;
            for(j = i+1; j < n; ++j)
            {
                x2 = P[j].x;
                y2 = P[j].y;
                d = sqrt((x1 - x2)^2 + (y1 - y2)^2);
                if(d < dmin)
                {
                    dmin = d;
                    index[0] = i;
                    index[1] = j;
                }
            }
        }

        if(dmin == INT32_MAX)
            return NULL;
        return index;
    }
```

算法分析:

- 该算法的基本操作是计算两个点的欧几里得距离。该操作中的平方根计算在计算机中只能近似求解，而且对于计算机而言不是一件轻松的工作。所以应该避免求平方根。
- 基本操作的执行次数为 $\Theta(n^2)$。

6 凸包问题

凸集合的定义：对于平面上的一个点集合，如果以集合中任意两点 P 和 Q 为端点的线段都属于该集合，那么这个集合就是凸集合。

凸包的定义：对于平面上 n 个点的集合，它的凸包就是包含所有这些点的最小凸多边形。

凸包问题：为平面上 n 个点的集合构造凸包。

极点的定义：将最小凸多边形的顶点称为极点。

算法思路：对于一个 n 个点集合中的两个点 P_i 和 P_j，这两个点连成一条直线 l，当且仅当该集合中的其他点都位于这条直线 l 的同一边时，l 是该集合凸包边界的一部分。对每一对点都做一遍检验之后，满足条件的线段就构成了该凸包的边界。

算法实现:

```
    // 两个点连成一条直线，方程为ax+by=c
    // 一条直线把平面分为两个半平面，其中一个半平面上的点都满足ax+by>c，而另一个半平面中的点都满足ax+by<c
    // 为了检验某些点是否位于这条直线的同一边，可以简单地把每个点代入ax+by−c，检验这个表达式的符号是否相同
```

```c
struct Point
{
    double x;
    double y;
}
typedef struct Point Point;
Point* solution(Point* p, int len)
{
    Point* result = (Point*)malloc(len * sizeof(Point));
    int i,j,z;
    double a,b,c;
    double temp_k,temp_b;
    double x1, x2, x3;
    double y1, y2, y3;
    int flag[2];
    int index = 0;
    for(i = 0; i < len; ++i)
    {
        result[i].x = 0;
        result[i].y = 0;
    }
    for(i = 0; i < len; ++i)
    {
        x1 = p[i].x;
        y1 = p[i].y;
        for(j = i + 1; j < len; ++j)
        {
            flag[0] = 0;
            flag[1] = 0;
            x2 = p[j].x;
            y2 = p[j].y;

            if(x1 == x2)
            {
                b = 0;
                c = 1;
                a = 1/x1;
            }
            else
            {
                temp_k = (y1-y2)/(x1-x2);
                temp_b = y1 - temp_k*x1;
                a = temp_k;
                b = -1;
                c = -temp_b;
            }

            for(z = 0; z < len; ++z)
            {
                if(z != i && z != j)
                {
                    x3 = p[z].x;
                    y3 = p[z].y;

                    if(flag[0] == 0 && a*x3 + b*y3 > c)
                        flag[0] = 1;
```

```
                    else if(flag[1] == 0 && a*x3 + b*y3 < c)
                        flag[1] = 1;
                }
            }

            if(flag[0] && flag[1])
                continue;
            result[index++] = p[i];
            result[index++] = p[j];
        }
    }
}
```

算法分析：时间复杂度为 $\mathcal{O}(n^3)$。

7 穷举查找

7.1 旅行商问题

旅行商问题：要求找出一条 n 个给定的城市间的最短路径，这条最短路径的出发城市和终点城市是一样的。而且要求我们在回到出发的城市之前，对每个城市都只访问过一次。

算法思路：通过生成 n-1 个中间城市的组合来得到所有的旅游路线，计算这些线路的长度，然后求得最短的线路。

算法分析：算法时间复杂度为 $\Theta((n-1)!)$。

7.2 背包问题

背包问题：给定 n 个重量为 $w_1, w_2, ...w_n$，价值为 $v_1, v_2, ...v_n$ 的物品和一个承重为 W 的背包，求这些物品中一个最有价值的子集。

算法思路：计算出每个子集的总重量，找出所有可行的子集，然后在它们中间找到价值最大的子集。

算法分析：一个 n 元素集合的子集数量为 2^n，所以算法的时间复杂度为 $\Omega(2^n)$。

7.3 分配问题

分配问题：有 n 个任务需要分配给 n 个人执行，一个任务一个人。将第 j 个任务分配给第 i 个人的成本为 C[i,j]。要求找出总成本最小的分配方案。

算法思路：生成整数 1,2,...,n 的全部排列，然后把成本矩阵中的相应元素相加来求得每种分配方案的总成本，最后选出其中具有最小和的方案。

算法分析：算法时间复杂度为 $\Theta(n!)$。