算法专题：分治法（Divide and Conquer）

一：分治法概述

分治法思想

分治法的基本思想是将问题分成（divide）多个子问题，再递归（Conquer）的解决每个子问题，再将子问题的解合并（Combine）成原问题的解。

1）*这个思想是不是很熟悉，第一反应能不能想到**归并排序**和**快速排序**。后面我们会对两种排序和分治法的思想进行具体分析。*

2）分成多个子问题，有没有想到动态规划和贪心法，它们有什么区别呢？

看一下这三者的主要思想：

> - **分治法**：将问题分解成多个子问题，并允许不断分解，使规模越来越小，最终可用已知的方法求解足够小的问题。使用要求：（1） 问题能够按照某种方式分解成若干个规模较小、相互独立且与原问题类型相同的子问题。（2） 子问题足够小时可以直接求解。（3） 能够将子问题的解组合成原问题的解。1
> - **贪心法**：总是选择当前最优解，并不考虑整体最优。拥有最优子结构特性。当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。1
> - **动态规划法**：将较大问题分解为较小的同类子问题，这一点上它与分治法和贪心法类似。动态规划法解决子问题重叠现象，利用最优子结构，自底向上从子问题的最优解逐步构造出整个问题的最优解。适用动态规划法的要求：具有最优子结构特性和重叠子问题。与贪心算法比较：贪心算法局部的最优性依赖于其前面各部分是否最优；且不能保证最终解的最优性。动态规划当前决策的最优性取决于其后续决策系列的是否最优。动态规划方法可以保证问题求解是全局最优的。1
>
> （来源：new bing。其中答案点击可达相应链接）

分治法的基本步骤：

```
divide-and-conquer(P)
{
  if ( |P| <= n0) adhoc(P);   //解决小规模的问题
  divide P into smaller subinstances P1,P2,...,Pk; //分解问题
  for (i=1,i<=k,i++)
  yi=divide-and-conquer(Pi);  //递归的解各子问题
  return merge(y1,...,yk);  //将各子问题的解合并为原问题的解
}
```

这个步骤为什么是这样子呢？

这个来源于递归的函数的运行轨迹。

先解决小规模问题指的是不用进行递归就能解决的极小规模问题，比如求一个数组的最大最小值，数组只有一个和只有两个的情况完全不需要递归就可以直接解决返回。

然后将问题先分解，进行数次递归分解到最后然后从最后的递归依次往下执行接下来合并子问题的解的代码，回到上一层稍大问题的递归，再执行下面合并子问题的解的代码……直到回到原先要解决的大问题，答案搞定。

这个过程其实是可以画出一个递归树，因此分治法很好的将n的时间复杂度降低到了logn。

二：分治法的应用实例

求一组数的最大值和最小值

> 这个是指找出一组数中最大值和最小值的在数组中的位置。若数组大小为n。

思路一：（非递归1）分别设立min，max为数组的第一个数，然后一起与后面所有数进行比较，需要进行2*（n-1）次比较。

```
min←a[0];max←a[0];//伪代码
For i←1 to n-1 do
    if a[i]<min
            min←a[i]
       else if a[i]>max max←a[i]
```

思路二：（递归）将一组数分为（1）1-n/2和（2）n/2+1-n两部分，再将（1）和（2）分别分为两部分，依次往下分解，知道分成单独的一个或两个数，进行比较，设为最大和最小，再依次向上得到最大值。

```
Max-min(A[0,n-1],max,min)//伪代码
  if n=1 max←min←a[0],return;//解决小规模问题
  if n=2 {
      if a[0]≥a[1] max←a[0],min←a[1] //解决小规模问题
              else max←a[1],min←a[0];
      } else m←n/2
            Max-min(A[0,m-1],max1,min1),Max-min(A[m,n-1],max2,min2),//递归的解决各类子问题
            max←max(max1,max2), min←min(min1,min2),//将子问题的解合并为原问题的解
            return.
```

思路三：（非递归2）这个是对非递归的一点优化。

假如是偶数的话，可以两个一组的进行比较，如果是奇数个，先将第一个当作最大最小值，再先将二三比较依次，将其中较大的与前面较大的比较，较小的同理，最终得到结果。次数降低为[3*n/2]-2次，有了一定的优化效果。

大整数乘法

> 大整数乘法是指两个超过计算机整数类型范围的整数相乘。分治法是解决大整数乘法问题的有效方法。对于大整数乘法，我们可以将两个大整数分别拆分成高位和低位两部分，然后使用分治法递归地计算高位和低位的乘积，最后将结果合并起来即可。

具体来说，我们可以将两个大整数X和Y分别拆分成A、B、C、D四个部分，然后按照以下公式计算它们的乘积：

$$XY = AC * 2^n + [(A - B) * (D - C) + AC + BD] * 2^(n/2) + BD$$

其中n表示X和Y的位数，AC、BD表示A、C和B、D的乘积，(A-B)*(D-C)表示交叉项的乘积。这样，我们就将一个规模为n的大整数乘法问题转化为了4个规模为n/2的子问题，可以使用递归方法求解。同时进行复杂度分析T(n)=3T(n/2)+O(n);因此该算法的时间复杂度为O(n^log2(3))。在实际应用中，该算法比传统方法更加高效。

注：这个为什么是3倍的T(n/2)我想了挺久，后来发现是对时间复杂度和主方法的理解模糊导致的。如果

$$XY = AC * 2^n + (AD + BC) * 2^(n/2) + BD;$$

的话，T(n)=4T(n/2)+O(n)；时间复杂度就成了O(n^2)，这个三倍的T（n/2)和四倍的主要在于XY中乘法操作的次数，因为在对两个大整数进行相乘时，也是先递归往下分成小的子问题，在四倍时，是分成了求AC，AD，BC，BD四个子问题，而在三倍时，分成了求AC，BD，以及（A-B）*（D-C）的三个子问题。这时候看分成几个子问题采用了基础操作计数法的思想。

> 辨：基础操作计数法和渐进时间复杂度
>
> 基本操作计数法和渐进时间复杂度都是用来判断算法时间复杂度，但思路和应用场景不同。
>
> **基本操作计数法**是通过计算算法中基本操作执行的次数来估算时间复杂度。基本操作是指算法中执行次数最多的操作，通常是最内层循环的循环体。这种方法适用于**顺序结构和循环结构**的算法。
>
> **渐进时间复杂度**是指当输入规模趋近于无穷大时，算法的时间复杂度的增长速度。通常使用大O记号来表示渐进时间复杂度，例如O(n)、O(nlogn)、O(n^2)等。其中，n表示输入规模，logn表示以2为底的对数。**渐进时间复杂度关注算法的增长趋势，不关注具体的执行时间**。在实际应用中，我们通常使用渐进时间复杂度来评估算法的效率和优劣。
>
> 基本操作计数法适用于顺序结构和循环结构的算法，而渐进时间复杂度适用于各种类型的算法。在实际应用中，我们可以根据具体情况选择合适的方法来进行分析。
>
> 因此，基本操作计数法评出的时间复杂度和渐进时间复杂度之间没有直接关系。但是，在实际应用中，我们可以通过基本操作计数法评估出一个算法在特定输入规模下的执行效率，并根据其增长趋势来推断其渐进时间复杂度。

矩阵乘法也和这个大整数乘法问题的思路相似，都是通过更改小问题规模的个数来降低时间复杂度。

归并排序和快速排序

> 这两个排序方法都属于分治法，我们来看看这两个经典排序方法是如何运行分治法的，它们的差别在哪里。分治法不断利用递归往下分组的方法建议想成一个递归树。

归并排序思想：将待排序数组分成两个子数组，将每个子数组再分别分成两个数组，一直往下分直到不能分为止，由于每两个数组都是由一个数组分出来的，所以再利用辅助数组将两个数组重新合并，并且在合并的过程排好序（类似双指针算法）。也就是一个递归树，从最大的数组不断分到底层的子数组，子数组中只有一个数时，两两合并同时排序向上走，走到最上面时就已经排好序了。

快速排序思想：先选择一个基准元素，将待排序数组划分为左右两个子数组，左边子数组中所有元素都小于基准元素，右边子数组中所有元素都大于基准元素。然后对左右子数组同样选择一个基准玄素，将子数组中小于基准元素放左边数组，大于放右边数组，依次往下，到达递归树的最底层时，顺序已经排好。

这样子看其实归并排序更接近与我们1中给出的分治法的模板，先分组，在处理。而快速排序是先处理，再分组。

来看看两种排序的代码：

```
//归并排序
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;

    merge_sort(q, l, mid), merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];
    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}

//快速排序
void quick_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[l + r >> 1];
    while (i < j)
    {
        do i ++ ; while (q[i] < x);
        do j -- ; while (q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }

    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}

作者：yxc
链接：https://www.acwing.com/activity/content/code/content/39790/
来源：AcWing
```

这两种分治法排序的思想和代码已经实现了，然后就来分析一下时间复杂度。

对于归并排序，正常分成k份时，T(n)=t(n/k)+t(n-n/k)+cn；分成两份时最好T（n）=2*T（n/2）+cn，渐进时间复杂度为O（nlogn）

对于快速排序，由于这个时找基准元素，小的在左边，大的在右边，所以基准元素的选择和时间复杂度有着很大的关系，想象一下最差的T(n)=T(1)+T(n-1)+O(n)，我们每次选的基准元素是最小值或者最大值，那就需要O（n^2)的时间复杂度，是一件很糟糕的事情，最好的话当然是选择一个中间数作为基准点，T（n）=2*T（n/2）+O（n），时间复杂度最好就能到O（nlogn）了，这个也是和递归树的高度有关的。

> 那为什么其实最常用的是快速排序呢?

> 理由如下：
>
> 1. **速度快**：快速排序的时间复杂度为O(nlogn)，这意味着它可以在很短的时间内对大量数据进行排序。相比之下，其他排序算法如堆排序和归并排序的时间复杂度也为O(nlogn)，但是快速排序通常比它们更快，因为它可以就地操作，而不需要创建任何辅助数组来保存临时值。
> 2. **占用资源少**：快速排序使用的空间复杂度为O(logn)，这意味着它在内存占用方面比其他排序算法更加高效。
> 3. **局部性好**：快速排序中的分区步骤通常具有很好的局部性，因为它访问前后附近的连续数组元素。这使得计算机硬件可以通过优化访问位置来提高效率。

归并排序和逆序对

> 逆序：给定自然数 1, ⋯ , n 的一个排列，如果 j > i 但 j 排 在 i 的前面则称 (j, i) 为该排列的一个逆序。

分为两部分，求内部逆序数，求两个之间的逆序数。

```
void MergeSort(int r[], int low, int high) { // 进行归并排序
    int mid, r1[1000], i;
    if (low == high)
        return;
    else {
        mid = (low + high) / 2; // 划分

        MergeSort(r, low, mid); // 递归求左子序列中逆序对
        MergeSort(r, mid + 1, high); // 递归求右子序列中逆序对

        Merge(r, r1, low, mid, high); // 合并
        for (i = low; i <= high; i++)
            r[i] = r1[i];
    }
}

void Merge(int r[], int r1[], int low, int mid, int high) { // 合并子序列
    int i = low, j = mid + 1, k = low;

    while (i <= mid && j <= high) {
        if (r[i] <= r[j]) {// 取较小者放入r1[k]中
            r1[k++] = r[i++];
        } else {
            count += mid - i + 1;
            // 若左子序列中的数1大于右子序列的数2，则数1后面的数都大于数2
            r1[k++] = r[j++];
        }
    }
    while (i <= mid) r1[k++] = r[i++];
    while (j <= high) r1[k++] = r[j++];
}
```

二分搜索问题

> 给定已按升序拍好序的n个元素a[0:n-1]，现要在这n个元素中找到一个特定的元素x。

遍历直接查找的时间复杂度为O(n)。

分治可将时间复杂度降低到O(lgn)。

伪代码如下：

```
template<class Type>
int BinarySearch(Type a[],const Type&x,int l,int x)
{
    while(r>=l)
    {
        int m=(l+r)/2;
        if(x==a[m])return m;
        if(x<a[m])r=m-1;else l=m+1;
    }
}
```

选择问题

> 寻找数组中第k小的元素。

1）先排序，再可以直接找到第k小的元素。

2）利用快速排序的方法，筛选出比基准元素小的和比基准元素大的，再根据左右边的数量判断k在哪一边，按照这个思路往下推，直到找到为止。

对于1）的复杂度分析就略过了，这里重点看看2）

当左右大小相差不多时，可以得到以下表达式：T（n）<= T（n/2）+ cn，n为2的幂次方的情况下，时间复杂度较好能达到O(n)，当然基准元素最好是要能选到中间的。代码如下。

```
T select (T a[], int l, int r, int k) {
    if (l >= r)return a[l];
    int i = l, j = r + 1;
    T pivot = a[j];

    while (true) {
        do {
            i++;
        } while (a[i] < pivot);
        do {
            j--;
        } while (a[j] > pivot);
        if (i >= j)break;
        swap(a[i], a[j]);
    }
    if (j - l + 1 == k)return pivot;

    a[l] = a[j];
    a[j] = pivot;
```

```
    if (i - l + 1 < k)
        return select(a, j + 1, r, k - j + l - 1);
    else return select(a, l, j - 1, k);
}
```

棋盘覆盖问题

> 在一个2的k次方乘2的k次方个放个组成的棋盘上，恰有一个方格和其它方格不同，称该方格为一特殊方格，且称该棋盘为一特殊棋盘。在棋盘覆盖问题中，要用4种不同形态的L型骨牌覆盖给定的特殊棋盘上的特殊方格以外的所有方格，且任何两个L型骨牌不能重叠覆盖。

递归式：T(k)=4T(k-1)+O(1);——>T(n)=4T(n/4)+c=O(n);

本来只有一个位置是特殊位置，每次切分时，加上一个骨牌，将四个位置都变成特殊位置。

```
void chessBoard(int tr, int tc, int dr, int dc, int size) {
    //dr,dc残缺方块所在行列
    //tr,tc棋盘中左上角方格所在行列
    if (size == 1) return;
    int t = tile++, // L型骨牌个数
        s = size / 2; // 子棋盘大小

    // 覆盖左上角子棋盘
    if (dr < tr + s && dc < tc + s)
        // 特殊方格在此子棋盘中
        chessBoard(tr, tc, dr, dc, s);
    else {// 此子棋盘中无特殊方格
        // 用 L型骨牌覆盖右下角
        board[tr + s - 1][tc + s - 1] = t;
        // 覆盖其余方格
        chessBoard(tr, tc, tr + s - 1, tc + s - 1, s);
    }

    // 覆盖右上角子棋盘
    if (dr < tr + s && dc >= tc + s)
        // 特殊方格在此子棋盘中
        chessBoard(tr, tc + s, dr, dc, s);
    else {// 此子棋盘中无特殊方格
        // 用L型骨牌覆盖左下角
        board[tr + s - 1][tc + s] = t;
        // 覆盖其余方格
        chessBoard(tr, tc + s, tr + s - 1, tc + s, s);
    }

    // 覆盖左下角子棋盘
    if (dr >= tr + s && dc < tc + s)
        // 特殊方格在此子棋盘中
        chessBoard(tr + s, tc, dr, dc, s);
    else {// 用 L型骨牌覆盖右上角
        board[tr + s][tc + s - 1] = t;
        // 覆盖其余方格
        chessBoard(tr + s, tc, tr + s, tc + s - 1, s);
```

```
    }

    // 覆盖右下角子棋盘
    if (dr >= tr + s && dc >= tc + s)
        // 特殊方格在此子棋盘中
        chessBoard(tr + s, tc + s, dr, dc, s);
    else {// 用 L型骨牌覆盖左上角
        board[tr + s][tc + s] = t;
        // 覆盖其余方格
        chessBoard(tr + s, tc + s, tr + s, tc + s, s);
    }
}

void OutputBoard(int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++)
            cout << setw (5) << Board[i][j];
        cout << endl;
    }
}
```

循环赛日程表

> 有n=2k个运动员进行网球循环赛，现在要设计一个日程表，满足以下要求：
>
> 1）每个选手必须和其它个选手各比赛一次。
>
> 2）每个选手一天只能赛一次。
>
> 3）循环赛一共进行n-1天。

按照分治策略，所有的选手分成两半的情况下，n个选手的日程表由外n/2个选手就可以决定。

代码如下：

```
void gametable(int k) {
    int a[100][100];
    int n, temp, i, j, p, t;

    n = 2; //两个参赛选手日程
    a[1][1] = 1;
    a[1][2] = 2;
    a[2][1] = 2;
    a[2][2] = 1;

    for (t = 1; t < k; t++)
        //迭代处理，依次处理2 ^ n....2 ^ k个选手的比赛日程
    {
        temp = n;
        n = n * 2; //填左下角元素
        for (i = temp + 1; i <= n; i++)
            for (j = 1; j <= temp; j++)
                a[i][j] = a[i - temp][j] + temp;

        //左下角和左上角元素的对应关系
```

```
        for (i = 1; i <= temp; i++)
            //将左下角元素抄到右上角
            for (j = temp + 1; j <= n; j++)
                a[i][j] = a[i + temp][(j + temp) % n];

        for (i = temp + 1; i <= n; i++)
            //将左上角元素抄到右下角
            for (j = temp + 1; j <= n; j++)
                a[i][j] = a[i - temp][j - temp];
    }
}
```

附：

1.内容参考

佟鑫宇老师 天津大学智能与计算学部 2023秋 算法设计与分析 ppt

2.

考试考察重点：棋盘覆盖问题，归并排序，快速排序，选择问题。

3.

全文同样包含个人的主观理解，如有错误，欢迎访问原文链接指正。