算法分析与设计2

1 算法核心

算法核心主要描述了面对问题的解决方法和核心思路，面对问题可以尝试使用不同的算法核心去匹配解决方案。

1.1 蛮力法

核心：找出所有可能，对于所有可能做出判断。

经典算法和问题：

- 选择排序：选择最小的元素放置到数组前面。
- 冒泡排序：选择最大的元素放置到数组后面。
- 顺序查找：遍历查找元素。
- 蛮力字符串匹配：从S1头部开始匹配S2，匹配失败进入S1下一位，S2重置。
- 凸包问题（找到多个“点对”之中最外层的点）：获取两个点，查看其它点是否在“这两点形成的连线”的同一侧。
- 旅行商问题、背包问题、任务分配问题：穷举所有可能结果，获取满足需求的结果。
- 深度优先算法、广度优先算法：针对某种排列的穷举方式。

1.2 减治法

核心：减少规模。

经典算法和问题：

1. 规模为**“n”**的问题**->**规模为**“n-1”**的问题

- 插入排序：遍历n次->（前i-1个都已有序，将i插入到有序中）
- 拓扑排序：遍历n次->（获取入度为0的点，删除它，放入结果中）
- 组合问题：递归n次->遍历n-1次->（选择n-1个剩余的点放入结果中）
- 生成子集：递归n次->遍历2次->（每个点都选择：1. 放入；2. 不放入）

2. 规模为**“n”**的问题**->**规模为**“n/a”**的问题。

- 折半查找：递归->（比对中间值，小于从左边找，大于从右边找）
- 假币问题（多个金币存在一个银币）：递归->（金币分为两/三堆，比较重量，选择轻的）
- 约瑟夫斯问题（“奇数人”杀死旁边的人）：递归->（杀死从小到大的偶数位的人）

3. 规模为**“n”**的问题**->**规模为**“<n”**的问题。

- 选择第k小元素：递归->（选择一个元素作为基准，小于基准的元素移动左侧；大于基准的元素移动右侧，则可以知晓基准的大小排名）

- 插值查找：递归->（认为有序是按照一定比例递进的，计算“low-high”的斜率，根据斜率得到比较下标）

1.3 分治法

核心：

- 将一个问题划分为同一类型的若干子问题，子问题规模尽量相同。
- 对子问题进行求解。
- 若有必要，合并子问题的解。

个人理解：

- 本质上，只是将问题分化为子问题，总体来看，仍然做了一定的遍历。
- 主要利用某种规律，将规模划分为相同的部分，再对该部分进行分治。

经典算法和问题：

- 归并排序：数组分为两部分，都进行排序；组合两个排序好的数组。
- 快速排序：选择一个元素作为基准，小于基准的元素移动左侧；大于基准的元素移动右侧。对小于部分进行排序；对大于部分进行排序。
- 二叉树：对某一个树结点的控制，可以转化为“对该结点的控制”和“该结点的左/右子节点的控制”，控制的执行顺序可以交换。

1.4 变治法

核心：将问题的解决分为多个子步骤来解决，每个子步骤负责不同部分；利用数据结构来解决问题。

经典算法和数据结构：

- 预排序：解决问题时，提前将元素进行排序。
- 平衡查找树：查找树的升级版，每个结点都有一个变量记录左子树和右子树的结点差，每次操作都会平衡结点差为-1,1,0。
- 堆：完全二叉树，从非叶子结点构建树，保证当前结点一定大于子节点。

1.5 动态规划

核心：导出一个问题实例的递推关系，该递推关系包含该问题的更小子实例的解。

个人理解：

- 记忆化分治法：子规模的可能会被重复求解，使用数组记录结果，直接从数组中获取；本质上，仍然做了一定的遍历。
- 原本的递归的变量在二维数组中成为了下标；递归方式变为了递推关系；递归的边界值（例如n = 1或n = 0时）需要在dp数组中初始化。
- 例如斐波那契，分治法：f(n) = f(n-1) + f(n-2)；动态规划：dp[n] = dp[n-1] + dp[n-2]。

经典算法和问题：

- 小偷问题（一排可偷的居民房，连续偷会触发警报，求偷到的价值最大化）：从2开始偷，偷到第i个房子的最大价值dp[i]为max(dp[i-1], dp[i-2] + v[i])。
- 阶梯问题（上阶梯，可以直接上1阶或2阶，求到n阶的走法个数）：从1阶开始上，到i阶的种类dp[i]为dp[i-1] + dp[i-2]。
- 背包问题：两个变量，使用二维动态规划。从第0个物品开始，遍历->（背包重量从0开始，到max的最大价值）。背包重量为j时，放入最大价值dp[i][j]为max(dp[i-1][j], dp[i-1][j-w])。

1.6 贪心

核心：可行性；局部最优->全局最优；每个决定不可取消。

经典算法：

- Prim算法：最小生成树。每次从“可延展边”中选择最小边。
- Krukal算法：最小生成树。每次“全局”选择的最小边，检查最小边的两个点是否联通，不联通则放入。
- Dijkstra算法：获取“所有结点”到“起始结点”的最短路径。从“可延展边”中选择最小边，更新“起始结点”到“延展点”的最小距离。

2 做题方法

蛮力法--是否可以按照某种规律减少问题的规模-->减治法

蛮力法--是否可以将问题分为不同部分，然后再解决-->分治法--是否可以使用记忆化来解决问题-->动态规划

蛮力法--蛮力的“局部选择最优”是否能够保证全局最优-->贪心

任何方法--是否可以利用预排序/某种数据结构来解决问题-->变治法

3 排序

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

- 冒泡排序：遍历n次->将最大的元素放置最后。
- 选择排序：遍历n次->选择最小的元素，放置指定位置。
- 插入排序：遍历n次->前n-1个元素都已经排好序，将第n个元素放置前方有序序列中。
- 希尔排序：执行logn次->将数组按照gap增量分组，gap从len/2开始到1，保证组内是有序的，使用合并排序，总共执行n次。
- 归并排序：执行logn次->将数组分为两组，分别将两组使用归并排序，得到两个有序数组，使用合并排序，总共执行n次。
- 快速排序：执行logn次->在数组内找到一个基准，小于基准的元素移动至左侧，大于基准的元素移动至右侧，小于基准的最后一个元素和基准元素交换位置；分别对小于部分和大于部分进行快速排序。
- 堆排序：1. 维护堆：从n/2~0开始，确保该元素大于自己的子结点，交换至最大子结点，聚焦于交换的结点，递归；2. 遍历n次->删除最大元素：将最大元素和最后一个元素交换，隔离交换后的最大元素，维护堆，继续删除。
- 计数排序：前提：保证数据能够“有序地”映射到数组的下标；1. 遍历n次->将数据映射的下标的数据+1，假设映射出k个；2. 遍历k次->从小到大，输出数据。
- 桶排序：设置k个桶，用于放入；1. 遍历n次->比较桶的大小，放入唯一满足条件的桶，桶内间维护有序；2. 按照桶的大小输出。
- 基数排序：前提：元素为数字类型；根据位数排序，从低位/高位开始；实现方式是桶排序（桶为数字0~9）。

4 算法模板

二分查找

1. 左闭右闭

```
public int search(int nums[], int size, int target) {
    int left = 0;
    int right = size - 1;
    while (left <= right) {
        int mid = (right + left) / 2;
        if (nums[mid] > target)
            right = mid - 1;
        else if (nums[mid] < target)
            left = mid + 1;
        else return mid;
    }
    return -1;
}
```

2. 左闭右开

```
public int search(int nums[], int size, int target) {
    int left = 0;
    int right = size - 1;
    while (left < right) {
        int mid = (right + left) / 2;
        if (nums[mid] > target)
            right = midlle;
        else if (nums[mid] < target)
            left = mid + 1;
        else return mid;
    }
    return -1;
}
```

希尔排序

- 按照gap增量分数组
- 子数组之间使用插入排序

```
public void shellSort(int[] arr) {
    int len = arr.length, tmp, j;
    for (int gap = len / 2; gap >= 1; gap = gap / 2) {
        // 分组执行插入排序
        for (int i = gap; i < len; i++) {
            tmp = arr[i];
            j = i - gap;
            while (j >= 0 && arr[j] > tmp) {
                arr[j + gap] = arr[j];
                j -= gap;
            }
            arr[j + gap] = tmp;
        }
    }
```

```
}
```

归并排序

- 二分数组
- 合并子数组

```
public void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
public void merge(int[] arr, int left, int mid, int right) {
    int[] tmp = new int[right - left + 1];
    int k = 0;
    int i = left, j = mid + 1;
    while (i <= mid && j <= right) {
        if (arr[i] < arr[j])
            tmp[k++] = arr[i++];
        else
            tmp[k++] = arr[j++];
    }
    while (i <= mid) tmp[k++] = arr[i++];
    while (j <= right) tmp[k++] = arr[j++];
    for (int t = 0; t < k; t++) {
        arr[left + t] = tmp[t];
    }
}
```

快速排序

- 基准
- 交换
- 子数组排序

```
public void quickSort(int[] arr, int left, int right){
    if (left > right) return;
    int i, j, temp, t;
    i = left + 1;
    j = right;
    temp = arr[left];
    while (i <= j) {
        while (temp <= arr[j] && i <= j) j--;
        while (temp >= arr[i] && i <= j) i++;
        if (i <= j) {
            t = arr[j];
```

```
            arr[j] = arr[i];
            arr[i] = t;
        }
    }
    arr[left] = arr[j];
    arr[j] = temp;
    quickSort(arr, left, j-1);
    quickSort(arr, j+1, right);
}
```