《算法设计与分析》第一次手写作业

2024.9.26 唐治江

1.Longest Balanced Substring

1. **建模**：

- **输入**：一个字符串`s`
- **输出**：符合条件的子字符串
- **思路**：

  考虑将其这个问题划分为若干个子问题，考虑到只要包含不平衡字符（在字符串中只出现大写或小写）的字符串一定是不平衡的，所以可以利用不平衡字符将字符串划分为若干个子字符串，然后递归处理这些子串：

  - 首先遍历整个字符串，找到所有不平衡的字符。这些不平衡字符将作为分割点。
  - 将字符串根据这些不平衡字符分割成若干个子串。
  - 递归地处理这些子串，返回其中最长的平衡子串。

2. **算法描述**：

- 定义函数`find_balanced_subs(s)`：
  - 如果字符串`s`长度为1，则返回0

- 定义`unbalanced_char_set`用于存储字符串中不平衡的字符
- 遍历字符串，动态维护`unbalanced_char_set`（对于字符`c`若其大写或小写存在于`unbalanced_char_set`，则将其剔除，否则加入到`unbalanced_char_set`），并记录不平衡字符的索引。
- 将字符串按照这些不平衡字符的索引进行分割，递归处理每个子串。
- 在所有子串中找到最长的平衡子串并返回。

3. **时间复杂性**：$O(nlogn)$，递归的次数取决于不平衡字符的个数，但都是对数级的，递归深度为 O(log n)，在每一层递归中需要线性扫描子字符串，所以时间复杂度可以看作是$O(nlogn)$。
4. **空间复杂性**：O(n)，递归栈深度与字符串长度有关。

2. Cutting Bamboo Poles

1. **建模**

- **输入**：
  - 一个数组（$L = [l_1, l_2, \ldots, l_n]$）表示每根竹竿的长度
  - 一个整数（m）表示所需的竹竿数量。
- **输出**：一个整数，表示可以得到的每根竹竿的最大长度。
- **思路：**

  可以尝试二分查找的求解。竹竿长度的搜索范围为$(0, max_len])$，使用二分查找，每次计算每个中间值（mid）时能切割出的竹竿数量，如果能得到至少（m）根竹竿，则尝试更长的长度；否则，尝试更短的长度。最终返回能够切割出的最大长度。。

2. **算法描述**

- 设置长度范围的边界，最小值为 0（无法取到），最大值为所有竹竿的最大长度 max_len，最终搜索范围为$(0, max_len])$。
- **二分查找**：

1. 定义区间的左右`left`（初始化为0）、`right`（初始化为max_len）
2. 计算中间值 `mid`（即当前检查的长度），$mid = \frac{left+right}{2}$
3. 遍历所有竹竿，将每根竹竿长度除以 `mid` （同一根竹竿可被切割多次），并累加得到的长度为`mid`的竹竿数量。
4. 如果可以得到至少 m 根竹竿，说明当前长度 `mid` 可行，尝试更长的竹竿长度（更新`left`=`mid`）；否则，尝试更短的竹竿长度（更新`right`=`mid`）。
5. 当最小边界非常接近最大边界时，返回最后的最大可行长度。

3. **时间复杂性**：$O(n\log(max\_len))$，二分查找的时间复杂度为$O(\log(max_len))$，其中max_len 是竹竿的最大长度。遍历所有竹竿的时间复杂度为$O(n)$。因此，整个算法的时间复杂度为$O(n\log(max_len))$
4. **空间复杂性**：该算法除了存储竹竿长度外，只使用了常数级的额外空间来存储变量，因此空间复杂度为 $O(1)$。

3.Multiple Calculations

1. **建模**

- **输入**：一个字符串 `s`，由数字和运算符（`+`、`-`、`*`）组成。
- **输出**：所有可能的计算结果。
- **思路:**

  通过递归计算给定字符串中数字和运算符的所有可能结果。首先，遍历字符串以找到运算符，并将其分为左右子表达式。并进行时间上的优化，对于每个子表达式，检查其是否已经计算过，如果是，则直接返回结果。否则，递归计算其结果，并通过当前运算符合并左右子表达式的结果。

2. **算法描述**

- 定义函数`compute(s)`：
  - 如果字符串只包含数字，直接返回该数字作为结果。

- 检查当前子表达式是否已经在 `memo` （用于存储子表达式的结果）中。如果是，直接返回存储的结果。
- 遍历字符串 `s`，找到所有运算符的位置。
- 对于每个运算符`o`，将字符串分成左右两部分：左边的子表达式`s_left`和右边的子表达式`s_right`。并输入`compute`递归计算。
- 对于左右子表达式的每一对结果，根据运算符`o`进行计算，并将结果存储在一个列表中。
- 字典 `memo` 来存储已经计算的子表达式的结果。
- 返回结果

3. **时间复杂性**：$O(n^2)$，其中 n 是操作数的数量。每个子表达式的计算结果只需存储一次，避免了重复计算。
4. **空间复杂性**：空间复杂度为 $O(n^2)$，主要用于存储计算结果的字典和递归栈的深度。

4.N-sum

1. **建模**

- **输入**：
  - 一个数组 $B[0,1,...,n-1]$ ，其中每个元素是一个整数，范围在 $[0,n]$ 之间，可以重复选取元素。
  - 整数 (m)，代表我们需要达到的目标和。
- **输出**： 输出一个布尔值，表示是否存在选取恰好n 个数，使得它们的和等于 m。
- **思路**：

  构造双变量生成函数，其中x^k 表示选取的和为 k，t^p 表示选取了p个元素。

  - 生成函数现在不仅要表示选取的和，还要表示选取的次数：$(1+t\cdot x^{B[i]}+t^2\cdot x^{2B[i]}+\cdots)=\frac{1}{1-t\cdot x^{B[i]}}$
    - t^k 表示选取了k个$B[i]$。

        - $x^{k \cdot B[i]}$表示和为$k \cdot B[i]$。
    - 总的生成函数$G(t,x)$：

$$G(t,x) = \prod_{i=1}^{n} \frac{1}{1 - t \cdot x^{B[i]}}$$

    通过 FFT（快速傅里叶变换）进行多项式卷积，累积生成函数的乘积，检查最终生成函数中$t^n \cdot x^m$ 的系数是否大于 0。如果 $t^n \cdot x^m$ 项的系数大于 0，则说明存在选取 n个数和为 m 的组合；否则不存在。

2. **算法描述**

- 初始化生成函数$G(t,x) = 1$，表示最初没有选取任何元素。
- 对于数组 B 中的每个元素$B[i]$：
    - 构造单个元素的生成函数为$\frac{1}{1-t \cdot x^{B[i]}}$，使用有限项表示几何级数展开到 x^m。
    - 通过多项式卷积将生成函数$G(t,x)$ 与当前生成函数相乘。
    - 保留生成函数中到x^m 的项，确保多项式长度不会超过所需的目标和m。
- 当遍历完所有元素后，检查生成函数中的$t^n \cdot x^m$项的系数。
    - 如果$t^n \cdot x^m$ 项系数大于 0，返回 True，表示存在解。
    - 否则，返回 False，表示不存在解。

3. **时间复杂性**：构造生成函数并进行多项式卷积的时间复杂度为 $O(n^2 \log n)$。其中 n 是数组 B 的长度，$\log n$ 来源于 FFT 的多项式乘法。
4. **空间复杂性**：每次多项式卷积运算中，最多保留$m * n$项，因此空间复杂度为$O(m * n)$，即$O(n^3)$（因为$m \leq n^2$）。

5. Ex. Unary Cubic Equation

1. 建模

- **输入**：一个三次方程 $ax^3+bx^2+cx+d=0$，其中(a,b,c,d)为实数系数。
- **输出**：满足条件的三次方程的三个不同实根，并按照升序排列，根的差至少为 1，且每个根保留两位小数。
- **思路**：通过二分法，在导数的两个极值点之间以及在无穷远处确定根的存在性，可以通过二分法精确找到方程的三个实根。
  - 确保找到的三个实根之间的绝对差值都大于等于 1。
  - 将三个实根按照升序排列，并输出每个实根，保留两位小数。

2. **算法描述：**

- 计算方程的导数：$f'(x)=3ax^2+2bx+c$，找到该导数的两个极值点 (x_1,x_2)。
- **二分查找：**
  - 在范围$[-100,x_1]$、$[x_1,x_2]$、$[x_2,100]$进行二分查找
  - 设置左边界$left=-100$，右边界为 $right=x_1$。

  计算中间点 $mid=\frac{left+right}{2}$。

  计算 $f(left)$ 和 $f(mid)$：

  - 如果 $f(left)\cdot f(mid)<0$，则在区间 $(left,mid)$ 中存在根。将右边界更新为 mid。
  - 否则，将左边界更新为 mid。

  重复此过程，直到左边界和右边界的差小于给定的容忍度。
- 检查根的差值是否满足大于等于 1。将根按照升序排列，输出保留两位小数的结果。

3. **时间复杂性：** $O(\log n)$，递归每次将区间缩小一半。

4. **空间复杂性：** $O(1)$

6. Ex. Distance

1. **建模**

- **输入**:
  - 两个整数数组 `arr1` 和 `arr2`，长度分别为n_1 和n_2 。
  - 一个整数d ，表示距离阈值。
- **输出**：满足条件的元素个数，即在 `arr1` 中有多少个元素满足：对于每个 `arr1[i]`，在 `arr2` 中没有任何元素 `arr2[j]` 使得 $|arr1[i] - arr2[j]| \leq d$ 。
- **思路**:
  - 我们可以利用分治法来加速比较的过程。首先将 `arr2` 数组排序，使用分治快速定位每个 `arr1[i]` 与 `arr2` 中元素的距离关系。

2. **算法描述**

- 排序： 将数组 `arr2` 排序，方便后续进行分支查找。
- 对于`arr1`的每个元素`arr1[i]`，由于 `arr2` 是有序的，我们只需要检查离 `arr1[i]` 最近的两个元素。使用二分法在 `arr2` 中找到最接近的元素 `arr2[j]`，并检查该元素的差值是否满足 $|arr1[i] - arr2[j]| > d$:
  - 定左指针 `left = 0` 和右指针 `right = n_2 - 1`，表示二分查找的范围是整个 `arr2`。
  - 开始二分查找:
    - 计算中间位置$mid = \left(\frac{left+right}{2}\right)$。
    - 比较 `arr2[mid]` 与 `arr1[i]` 的差值:
      - 如果 $|arr1[i] - arr2[mid]| \leq d$，此时最近的差值为零，直接返回 `False`。
      - 如果 `arr2[mid] < arr1[i]`，说明 `arr2[mid]` 在 `arr1[i]` 的左边，需要向右半部分查找更接近的元素，因此更新左指针 `left = mid + 1`。

        - 如果 `arr2[mid] > arr1[i]`，说明 `arr2[mid]` 在 `arr1[i]` 的右边，需要向左半部分查找更接近的元素，因此更新右指针 `right = mid - 1`。
    - 返回`True`
- 记录`True`的个数，返回结果

3. **时间复杂性**：$O(n_2 \log n_2 + n_1 \log n_2)$，排序 `arr2` 的时间复杂度为 $O(n_2 \log n_2)$，对于每个 `arr1[i]` 使用二分查找的复杂度为 $O(\log n_2)$，所以查找过程的复杂度为 $O(n_1 \log n_2)$，总体时间复杂度为$O(n_2 \log n_2 + n_1 \log n_2)$。

4. **空间复杂性**：$O(n_2)$，主要使用的额外空间为排序过程的空间和递归栈的空间。递归深度为 $O(\log n_1)$，排序占用 $O(n_2)$的空间。总的空间复杂度为 $O(n_2)$。