总论

2022年11月17日 19:05

算法设计与分析目标：

- 1、如何判断哪个算法更高效：已有是算法的选择
  - 需分析比较算法运行效率
- 2、如何设计正确高效的算法：新算法的提出
  - 需掌握算法设计的方法论

算法的由来

- 古代各类算法著作：几何原本、周髀算经等
- 现代：主要是算法和计算机的结合
  - 1936年，艾伦·图灵提出图灵机，建立了通用的计算机模型，刻画计算机的计算行为
    - 图灵为理论计算机科学与人工智能之父
  - 1946年，冯·诺依曼提出了存储程序原理，奠定了现代计算机基本结构
    - 诺伊曼为现代计算机之父

算法的定义

- 计算问题：给定数据输入，计算满足某种性质输出的问题
- 良定义：定义明确无歧义
- 算法：对于给定计算问题，算法是一系列良定义的计算步骤

算法的性质

- 有输入和输出
- 有穷性：算法必须在有限个计算步骤后终止
- 确定性：算法必须是没有歧义的，不产生二义性
- 可行性：可以机械的一步步执行基本操作

算法的表示

- 自然语言：贴近人类思维，易于理解主旨；可能产生歧义
- 编程语言：精准表达逻辑，没有歧义；不同编程语言语法存在差异，且过于关注算法实现细枝末节
- 伪代码：兼顾自然语言和编程语言两方面的优势
  - 伪代码是一种非正式的语言，关注算法本质，便于书写阅读
  - 移植编程语言书写形式作为基础和框架
  - 按照接近自然语言的形式表示算法过程

算法的分析

- 如何评价算法的性能？
  - 算法的运行时间和所占内存大小
  - 分别用时间复杂度和空间复杂度表示
- 分析的原则：
  - 1、归纳基本操作
    - 如赋值、比较、运算等
  - 2、统一机器性能

假设基本操作的代价均为1

统一机器性能后，算法运行时间依赖于问题输入规模与实例

相同输入规模，实例影响运行：如插入排序时，最好情况（数组升序）需n-1次，最坏情况（数组降序）则n(n-1)/2次

3、分析最坏情况

该情况下，运行时间仅依赖于输入规模，且这个时间用T(n)表示，n为输入规模

4、渐进分析法：忽略T(n)的系数和低阶项，仅关注（保留）最高阶项，用记号Θ表示

渐近记号	名称
$T(n) = \Theta(g(n))$	渐近紧确界
$T(n) = O(g(n))$	渐近上界
$T(n) = \Omega(g(n))$	渐近下界

例如选择排序所需要的步数为1.5n^2+0.5n-1，根据渐进分析，得出其T(n)为n^2

注意：最坏情况n(n-1)/2表示逻辑上来说需要这么多的交换次数；而T(n)或时间复杂度，表示计算机运行该排序算法时需要执行赋值、判断等的所有步数

O()表示渐进上界，即某式子的最大值（同样丢掉所有系数和低次项）

如果式子是波动的，则就表示最大值，如果是单调的，则丢掉式子的系数和低次项即可

- **O记号示例**
  - $\cos n = O(1)$
  - $\frac{n^2}{2} - 12n = O(n^2)$
  - $\log_7 n = \frac{\log_2 n}{\log_2 7} = O(\log_2 n) = O(\log n)$
  - $\sum_{i=1}^{n} \frac{1}{i}$ （假设n是2的整数幂）

$$= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \cdots + \frac{1}{n}$$

$$< \frac{1}{1} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \cdots + \frac{1}{\frac{n}{2}} + \frac{1}{n}$$

$$= \frac{1}{1} + 2 \cdot \left(\frac{1}{2}\right) + 4 \cdot \left(\frac{1}{4}\right) + 8 \cdot \left(\frac{1}{8}\right) + \cdots + \frac{n}{2}\left(\frac{1}{\frac{n}{2}}\right) + \frac{1}{n} = \frac{1}{n} + \sum_{j=0}^{\log n - 1} 1$$

$$= \log n + \frac{1}{n} = O(\log n)$$

渐进下界与之同理

算法运行时间称为算法的时间复杂度，通常使用渐进记号O表示，一般指渐进上界

前面几种排序方法，其时间复杂度均为O(n^2)

如果渐进上界和渐进下界相等，则存在渐进紧确界且与上界和下界相等

算法设计思路

2022年11月17日 21:23

分而治之

步骤：

1、分解原问题

采用递归的方法不断分解原问题，直到分解成最小的子问题

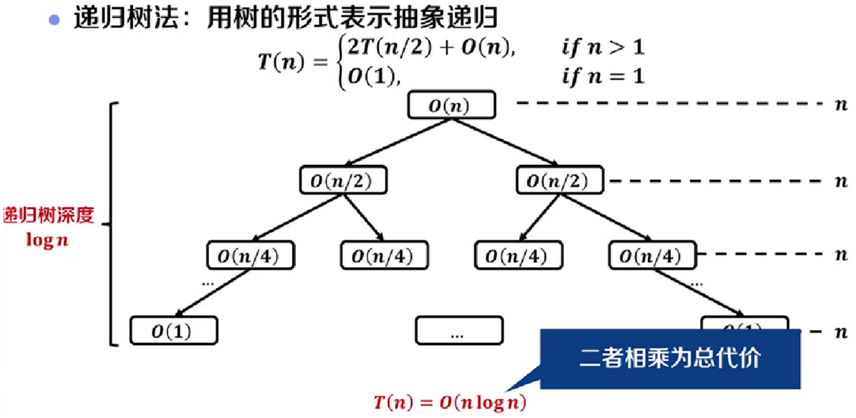
2、解决子问题

3、合并问题解：需要聚焦，因为这一步有时会直接影响算法效率

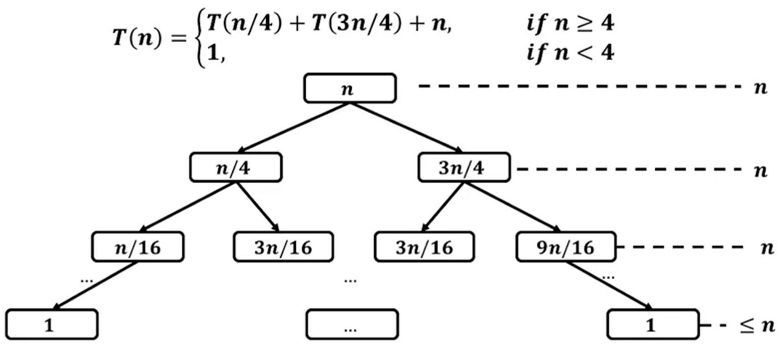
递归式求解：

递归树法：一般适合均分情况，可视化效果好

当有n个元素时，时间复杂度为：两个子问题的时间复杂度+合并问题解上确界O()，即一般形式为T(n)=T(n/b) + T((1-1/b)n)+ f(n)



如果遇到不均匀分配，如左节点树乘0.25，右子节点树乘0.75：



此时，深度为所有叶子节点中深度最大的那个（最坏分析），即一直乘0.75的那个

但该情况使用递归树法仍然不能确定渐进紧确界

对数换底公式：

$$\log_x N = \frac{\log_y N}{\log_y x}$$

代入法：先猜一个时间复杂度，然后代入并使用数学归纳法进行证明

- $T(n) = \begin{cases} T(n/4) + T(3n/4) + n, & n \geq 4 \\ 1, & n < 4 \end{cases}$
- 猜测：$T(n) = \Theta(n \log n)$
  - 即证明$\exists c_1, c_2, n_0 > 0$，使得$\forall n > n_0$，$c_1 \cdot n \log n \leq T(n) \leq c_2 \cdot n \log n$

问题：代入法不容易首次猜出时间复杂度

主定理法：对形如T(n) = aT(n/b) + f(n)的递归式

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

常见问题：归并排序：见排序算法

最大子数组问题：

从一个序列array中选出任意长度的连续一段，求这一段的所有元素之和达到最大时，这个序列的取法array[x:y]

分而治之：分割成两部分，分别计算两部分的最大数组，然后计算较小的那个数组和这两个数组之间的所有元素之和，看是否>0，如果>0，则最终结果为较大的+较小的+中间的元素，如果<0，则最终结果就是较大的那个

逆序对计数：计算数字序列中逆序的多少

动态规划（dynamic process）

记忆化搜索：其本质是若算法中存在重复计算的过程，则将其某一小步的计算结果保存而不用重复计算，下一次调用该计算时直接查值即可。是一种空间换时间的思想。动态规划就是一种优化后的记忆化搜索

求解斐波那契数列的递归函数为（求解F(50)，跑了十几分钟还没出结果）：

```
int fbnq(int n){
    if (n==1||n==2){
        return 1;
    }
    else{
        return fbnq(n-1)+fbnq(n-2);
    }
}
```

递归的形式虽然简洁，但存在大量重复计算，如：

F(7)=F(6)+F(5)，而F(6)=F(5)+F(4)，在计算时F(5)就被算了两次，越靠近这棵递归树的底部被重复计算的就越多，导致效率低下。若能在计算F(6)时，将其中间结果F(5)保存下来，则第二个F(5)就不必计算了，其它冗余计算同样处理，即可极大提高速度

若采用从前往后的循环算法，则可避免重复计算（计算F(50)，用时0.056s）：

```
def fbnq(n):
    if (n == 1) | (n == 2):
        return 1
    else:
        n1 = 1
```

```
        n2 = 1
        for i in range(n-2):
            n3 = n1+n2
            n1 = n2
            n2 = n3
        return n3
```

从前往后计算与使用动态规划的递归复杂度是相同的（使用字典保存）（计算F(50)，用时0.062s）：

```
dic = {}
def fbnq(n):
    if (n == 1) | (n == 2):
        return 1
    else:
        if str(n-1) in dic:
            a = dic[str(n-1)]
        else:
            a = fbnq(n-1)
            dic[str(n-1)] = a
        if str(n-2) in dic:
            b = dic[str(n-2)]
        else:
            b = fbnq(n-2)
            dic[str(n-2)] = b
        return a+b
```

多元动态规划问题：虽然比递归代码复杂，但效率高

```
# 1~n共n个位置，运动步数step=4，开始位置start=2，目标位置4
dic = {}
def step_num(start,aim,step,n):
    if step!=0:
        a=0
        b=0
        if start==1:
            if str(start+1)+'->'+str(aim)+'&'+str(step-1) in dic:
                b = dic[str(start+1)+'->'+str(aim)+'&'+str(step-1)]
            else:
                b = step_num(start+1,aim,step-1,n)
                dic[str(start+1)+'->'+str(aim)+'&'+str(step-1)] = b
        elif start==n:
            if str(start-1)+'->'+str(aim)+'&'+str(step-1) in dic:
                a = dic[str(start-1)+'->'+str(aim)+'&'+str(step-1)]
            else:
                a = step_num(start-1,aim,step-1,n)
                dic[str(start-1)+'->'+str(aim)+'&'+str(step-1)] = a
        else:
            if str(start+1)+'->'+str(aim)+'&'+str(step-1) in dic:
                b = dic[str(start+1)+'->'+str(aim)+'&'+str(step-1)]
            else:
                b = step_num(start+1,aim,step-1,n)
                dic[str(start+1)+'->'+str(aim)+'&'+str(step-1)] = b
            if str(start-1)+'->'+str(aim)+'&'+str(step-1) in dic:
```

```
                a = dic[str(start-1)+'->'+str(aim)+'&'+str(step-1)]
            else:
                a = step_num(start-1,aim,step-1,n)
                dic[str(start-1)+'->'+str(aim)+'&'+str(step-1)] = a
        return a+b
    else:
        if start==aim:
            return 1
        else:
            return 0
print(step_num(2,6,30,15))
```

除了用字典存储数据外，也可以用多元数组存储，并且同样可以优化数组的存储使其空间占用变小

动态规划表dp[]的作用，就是替代了递归的返回值，不用重复计算同样的问题，和上面的用字典存储其实本质上是一样的

贪心策略（greedy algorithm）

对于多步问题，每一步选择当下最好的路径（局部最优），不考虑以后的情况，并且希望这些局部最优也可以导致全局最优

例子：找零钱问题

描述：给顾客找零钱，需要使得给顾客的总钱币数最少

解决：可采用贪心算法，每次尽可能找最大面值的钱币

经典算法问题

2023年8月19日 14:43

摩尔投票算法：求一个序列中超过一半的个数的主元素

1. 时间复杂度：O(n)
2. 思想：对拼删除，从序列中选择两个不同的数字删除，最终剩下的一个或几个相同的数字一定是主元素（如果存在主元素）
3. 描述：遍历数组时维护一个候选元素和一个计数器。开始时候选元素为空，计数器为0。遍历数组，如果计数器为0，就将当前元素设为候选元素，然后如果当前元素和候选元素相同，计数器加1，不同就减1。这样，遍历完数组后，候选元素就可能是出现次数超过一半的主要元素。接着再遍历一次数组，统计候选元素出现的次数，确认是否超过一半，从而确定是否存在主要元素

两数之和：使用哈希表（字典）优化时间复杂度

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> hashtable;   // 创建哈希表
    for (int i = 0; i < nums.size(); i++) {
        auto it = hashtable.find(target - nums[i]);   // 在哈希表中查找目标值与
当前元素之差，auto可用作意外处理，find返回的是一个迭代器
        if (it != hashtable.end()) {   // 如果差值在哈希表中存在
            return {it->second, i};   // 返回差值的索引和当前元素的索引
}
        hashtable[nums[i]] = i;   // 若差值不存在，则将当前元素及其索引插入哈希表
}
    return {};   // 如果没有找到符合条件的两个数，则返回空向量
}
```

时间复杂度从N^2降低到N，空间复杂度从1上升到N

回溯算法：

采用试错的思想，尝试分步地去解决一个问题，本质上就是一个遍历

在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。回溯法通常用最简单的递归方法来实现

双指针法：将数组中的0全部移动到尾部且不改变其它元素相对位置

```
void moveZeroes(vector<int>& nums) {
        int n = nums.size(), left = 0, right = 0;
        while (right < n) {
            if (nums[right]) {
                swap(nums[left], nums[right]);
                left++;
            }
```

```
            right++;
        }
    }
```

双指针法：求解由数组元素和其距离所构成的能承载最多水量的容器

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int l = 0, r = height.size() - 1;
        int ans = 0;
        while (l < r) {
            int area = min(height[l], height[r]) * (r - l);
            ans = max(ans, area);
            if (height[l] <= height[r]) {
                ++l;
            }
            else {
                --r;
            }
        }
        return ans;
    }
};
```

装多少水由最短边决定。如果长边移动，装的水只可能会少，不可能会多，因为高度只可能小于等于短边，而宽度变小了。而如果移动最短边，那么有可能能够装更多的水。

哈希map+对字符排序，解决字母异位词

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string,vector<string>> mp;        //创建哈希表
        for (string& str:strs){     // 在strs容器里遍历每一个str元素，其类型为string&
            string key = str;       // 把str作为键
            sort(key.begin(),key.end());     // 对键排序，字母异位词排序后必然相同
            mp[key].emplace_back(str);      // 将str加入到key这个键所对应的vector里边
        }
        vector<vector<string>> ans;
        for (auto it = mp.begin();it!=mp.end();it++){    // 直接用auto，就懒得写复杂的迭代器类型了
            ans.emplace_back(it->second);       // it是对组，需用->访问值，即second
        }
        return ans;
    }
```

求某个vector中最长连续序列的长度

```
int longestConsecutive(vector<int>& nums) {
        unordered_set<int> num_set;
        for (const int& num:nums){    // 构造哈希表集合
            num_set.insert(num);
        }
        int longmax = 0;
        for (const int& num:num_set){
            if(!num_set.count(num-1)){      // 如果不存在num-1这个元素，则
count会返回0，即false
                int currentNum = num;
                int currentStreak = 1;
                while(num_set.count(currentNum+1)){
                    currentNum += 1;
                    currentStreak += 1;
                }
                longmax = max(longmax,currentStreak);   // 有多个可能的序
列，用longmax存储最长的那个
            }
        }
        return longmax;
    }
```

三数之和为0

```
// 位置不同，元素也不能相同
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(),nums.end());
        vector<vector<int>> ans;
        // 枚举a
        for (int first=0;first<n;first++){
            if (first>0&&nums[first]==nums[first-1]){
                continue;    // 保证与上一个枚举的数不相同
            }
            // c对应的指针初始指向数组的最右端
            int third = n-1;
            int target = -nums[first];  // 保证b+c=target
            for (int second=first+1;second<n;second++){
                if (second >first+1&&nums[second]==nums[second-1]){
                    continue;
                }
                while(second<third && nums[second]+nums[third]>target){
                    third--;
                }
                if (second==third){
                    break;      // 如果指针重合，随着b的增加，仍不会满足
a+b+c=0了，可直接退出
                }
                if (nums[second]+nums[third]==target){
                    ans.push_back({nums[first],nums[second],nums[third]});
                }
            }
        }
```

```
        return ans;
    }
```

异或运算的巧用：

```
int singleNumber(vector<int>& nums) {
        int single = 0;
        for (int num:nums){
            single ^= num;
        }
        return single;
    }
```

常见问题

2024年3月11日 21:59

在使用指针（及数组的伪指针）时，尽量不要全部使用`i`、`j`之类的，可以多使用`right`、`left`之类的指针，方便思考

为了保证计算子问题能够按照顺序、不重复地进行，动态规划要求已经求解的子问题不受后续阶段的影响。这个条件也被叫做「无后效性」。换言之，动态规划对状态空间的遍历构成一张有向无环图，遍历就是该有向无环图的一个拓扑序。有向无环图中的节点对应问题中的「状态」，图中的边则对应状态之间的「转移」，转移的选取就是动态规划中的「决策」。