数据结构与算法 I 思考题 8

2019201409 于倬浩

2020 年 12 月 5 日

16.2-6

将分数背包算法从 $\Theta(nlgn)$ 优化到 $\Theta(n)$ 的时间复杂度。

首先，改进的算法一定不能改变贪心算法的贪心策略，因此采用和 $\Theta(nlgn)$ 算法相同的贪心策略，显然不需要赘述原算法贪心策略的正确性。

考虑原算法的瓶颈，在于将所有物品按照平均值（价值/重量）从大到小排序，排序的过程需要 $\Theta(nlgn)$ 的复杂度，而贪心策略本身其实就是选择平均值最大的几个物品，且最后一个物品可能选到分数份。

因此，考虑使用之前学过的最坏时间复杂度 $\Theta(n)$ 的线性选择中位数算法 `select()` 进行优化，目的依旧是找出平均值最大的若干个物品来填满背包。

首先，对于子问题 `FractionalKnapsack(L, R, weight)`，表示只选择下标（可能被重标号）[L,R] 范围内的物品，且背包容量为 weight 的最优解。首先线性选出当前区间的平均值的中位数，统计平均值大于等于中位数的物品的重量之和。如果重量和小于当前的上限，说明最终的答案取到的最小平均值一定更小，因此解为子问题 `FractionalKnapsack(L, p - 1, weight - curweight)` 的解与这些平均值大于中位数的物品的并集；如果重量和大于等于当前的上限，说明最终答案取到的最小平均值更大，也就是这些物品的一个子集，因此为子问题 `FractionalKnapsack(p + 1, R, weight)` 的解。边界条件是当区间长度为 1 时，需要返回 `item[L].avg * min(weight, item[L].weight)`，表示最后一个物品可以选择分数份。

```
double FractionalKnapsack(int L, int R, double weight) {
    if(weight <= 0) return 0; //边界：背包已满
    if(L == R) return item[L].avg * min(weight, item[L].weight);
    //边界：需要切分的最后一个物品
    int p = select(L, R);
```

```
    //线性选择算法，找到区间 [L,R] 的中位数对应下标，且保证 [L,p] 都<= 中位数，[p+1,R]>= 中位数
    double curweight = 0, curvalue = 0;
    for(int i = p; i <= R; ++i) { //计算平均值大于中位数的物品的重量、价值之和
        curweight += item[i].weight;
        curvalue += item[i].value;
    }
    if(curweight <= weight) //含义如上
        return curvalue + FractionalKnapsack(L, p - 1, weight - curweight);
    else
        return FractionalKnapsack(p + 1, R, weight);
}
```