算法设计与分析实验2

计科2205 刘志垚 202208010512

1.用动态规划法实现0-1背包

问题描述

假设给定 n 个物体和一个背包，物体 i 的重量为 w_i，价值为 v_i（$i=1,2,......,n$），背包能容纳的物体重量为 c，要从这 n 个物体中选出若干件放入背包，使得放入物体的总重量小于等于 c，而总价值达到最大。

思路

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

状态转移:

假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品：

①$j<w[i]$，背包容量小于物体 i 重量

当前物体的重量大于背包容量，不能将该物体放入背包，总价值不变。即：

$$f_{i,j}=f_{i-1,j}$$

② $j\geq w[i]$，背包容量大于等于物体 i 重量

当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变。这种情况的最大价值是 $f_{i-1,j}$。

当其放入背包时，背包的容量会减少 w_i，背包中物品的价值增加 v_i，这种情况下的最大价值为 $f_{i-1,j-w_i}+v_i$。

由此可得状态转移方程：

$$f_{i,j}=max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

代码

```
//0-1背包
#include <bits/stdc++.h>
#define endl '\n'
typedef long long ll;
using namespace std;
const int N = 500005;
ll dp[N][N], v[N], w[N];
ll n, c;
void solve() {
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= c; ++j) {
            if(j < w[i])
                dp[i][j] = dp[i - 1][j];
            else
                dp[i][j] = max(
                    dp[i - 1][j - w[i]] + v[i],
                    dp[i-1][j]
```

```
                );
            }
        }
}
int main() {
    ifstream ifs("testData/test3000.txt");
    ofstream ofs("results/test3000.txt");
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    ifs >> n >> c;
    for(int i = 1; i <= n; ++i) ifs >> w[i] >> v[i];
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    solve();
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    ofs << dp[n][c] << endl;
    ofs << "time: " << duration.count() << " ms" << endl;
    // for(int i = 1; i <= n; ++i) {
    //     for(int j = 1; j <= c; ++j) {
    //         cout << dp[i][j] << " ";
    //     }
    //     cout << endl;
    // }
    return 0;
}
```

0-1背包空间优化

观察二维 DP 状态转移方程：

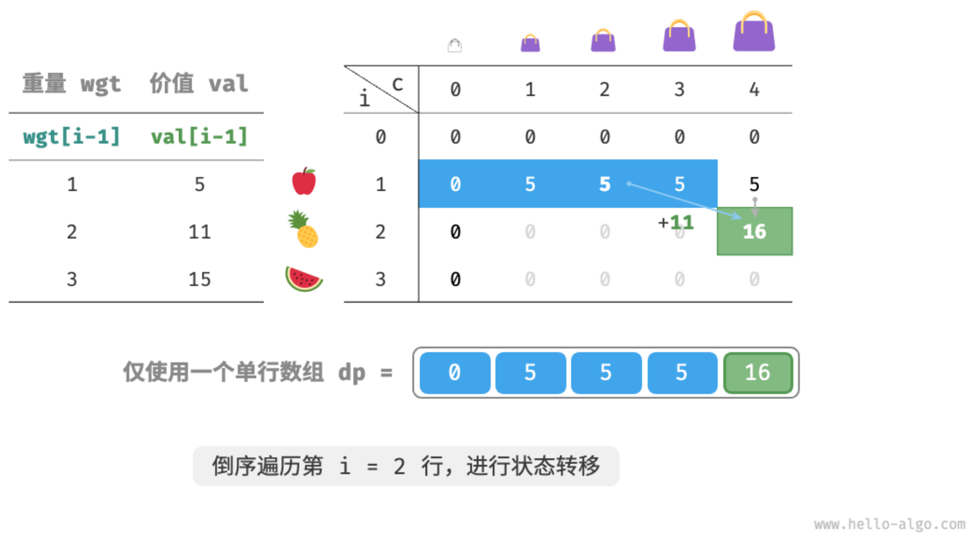
$$f_{i,j} = max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

由于每个状态都只与其上一行的状态有关，每个状态都是由`正上方`或`左上方`的状态转移过来的。因此我们可以可以去掉维度 i，使用数组滚动，将空间复杂度从 $O(n \times c)$ 降至 $O(c)$。

$$f_j = max(f_j, f_{j-w_i} + v_i)$$

注意：采用倒序遍历!!!这样才不会发生覆盖问题，状态转移可以正确进行。

Step 1



Step 2

Step 3

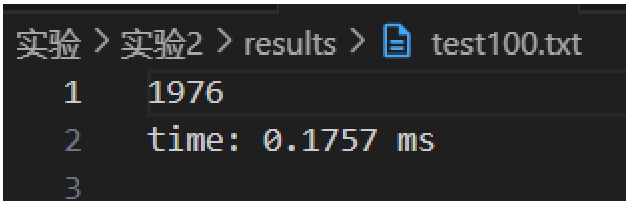
在代码实现中，我们仅需将数组 `dp` 的第一维 i 直接删除，并且把内循环更改为倒序遍历即可：

```
void solve() {
    for(int i = 1; i <= n; ++i) {
        for(int j = c; j >= w[i]; j--) {
            dp[j] = max(dp[j - w[i]] + v[i], dp[j]);
        }
    }
}
```

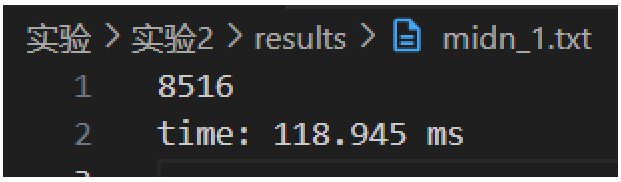
空间复杂度优化为 $O(c)$。

测试结果

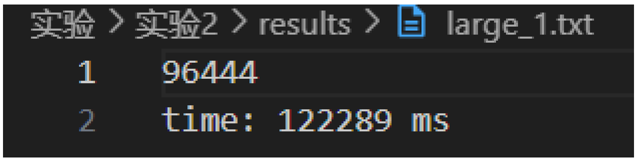
小规模：100个物品，背包容量为100



中规模：10000个物品，背包容量为5000



大规模：1000000个数据，背包容量为50000



算法分析

时间复杂度：$O(n \times c)$，显然，外循环 n 次，内循环 c 次。

空间复杂度：与 `dp` 数组有关。优化前 $O(n \times c)$。优化后 $O(c)$。

3.半数集问题(实现题2-3)

问题描述

2-3 半数集问题。

问题描述：给定一个自然数 n，由 n 开始可以依次产生半数集 set(n)中的数如下：

（1）$n\in$set(n)；

（2）在 n 的左边加上一个自然数，但该自然数不能超过最近添加的数的一半；

（3）按此规则进行处理，直到不能再添加自然数为止。

例如，set(6)={6, 16, 26, 126, 36, 136}。半数集 set(6)中有 6 个元素。注意，该半数集是多重集。

算法设计：对于给定的自然数 n，计算半数集 set(n)中的元素个数。

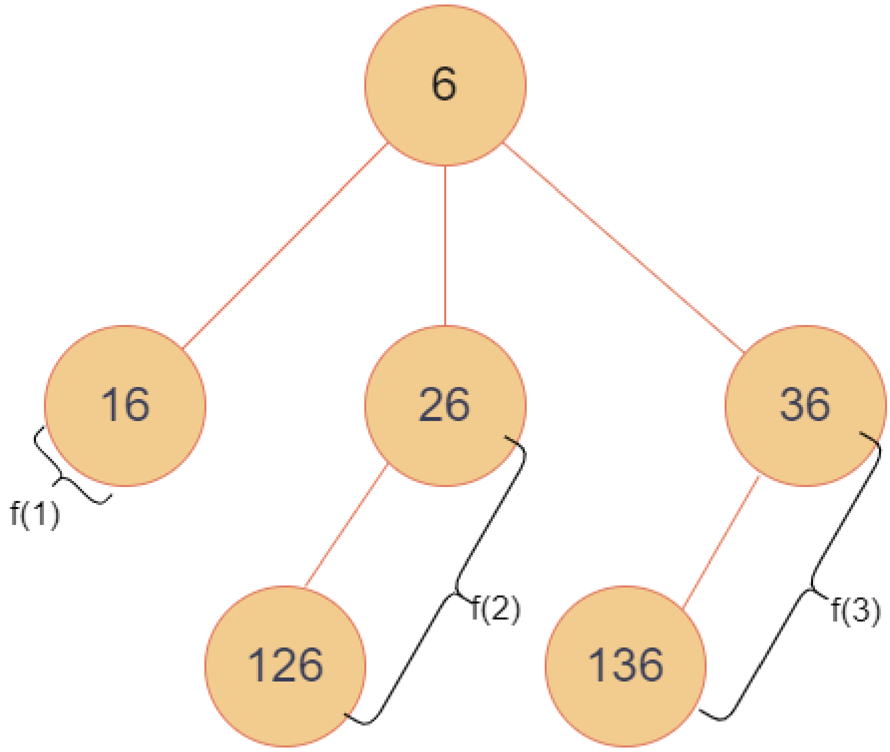
数据输入：输入数据由文件名为 input.txt 的文本文件提供。每个文件只有一行，给出整数 n（$0<n<1000$）。

结果输出：将计算结果输出到文件 output.txt。输出文件只有一行，给出半数集 set(n)中的元素个数。

输入文件示例	输出文件示例
input.txt	output.txt
6	6

思路

先来了解一下半数集是如何生成的，以set(6)为例：



我们找到了3个满足条件的数，即1、2、3，他们分别与6构成了16、26、36，依照这三个数继续向集合中添加元素，对于16，1是最近添加的数，但因为比1的一半小的自然数只有0，所以这一步就结束了。对于26，2是最近添加的数，2的一半是1，所以将126也添加到了半数集中。同理，将136也添加到了半数集中。

设set(n)的元素个数为 $f(n)$，经过以上例子不难发现，$f(6)=f(3)+f(2)+f(1)+1$。因此，有递归表达式：

$$f(n)=1+\sum_{i=1}^{n/2} f(i)$$

上述算法中显然有很多的重复子问题计算。用数组存储`已计算的结果`，避免重复计算，可明显改进算法的效率。

代码

```
// 实现题 2-3 半数集问题
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;
const int N = 50000005;
typedef long long ll;
int n;
ll f[N];
ll solution(int n) {
    ll sum = 1;
    if(f[n]) return f[n];
    for(int i = 1; i <= n/2; ++i) {
        sum += solution(i);
    }
    f[n] = sum;
    return sum;
}
int main() {
    ifstream ifs("testData/input2.txt");
    ofstream ofs("results/output2.txt");
    ifs >> n;
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    ofs << solution(n) << endl;
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    ofs << "time: " << duration.count() << " ms" << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

`关键代码及解释`：

```
void solution(int n) {
    ll sum = 1;
    if(f[n]) return; //如果f[n]不为0，表示已经计算过，直接返回
    for(int i = 1; i <= n/2; ++i) {
        solution(i);
        sum += f[i];
    }
    f[n] = sum; //存放结果
    return;
}
```

这里是递归的代码，这里使用 `记忆化数组f` 用来存放已经计算过的半数集，避免重复计算，提高算法效率。

测试结果

小规模：n=6

```
实验 > 实验2 > results > output2.txt
1    6
2    time: 0.0113 ms
```

中规模：n=10000

```
实验 > 实验2 > results > output2.txt
1    2220574865734494444
2    time: 9.2972 ms
```

大规模：n=1000000

```
实验 > 实验2 > results > output2.txt
1    8824746625766418600
2    time: 108531 ms
```

算法分析

时间复杂度：如果 `f` 数组在未被初始化的情况下存储先前的计算结果，可以减少重复计算，从而降低实际的运行时间。使用记忆化后，实际时间复杂度会降到 $O(n)$，因为每个 `f[i]` 只会计算一次。

空间复杂度：$O(n)$。取决于数组 `f[n]`。

4.集合划分问题(实现题2-7)

问题描述

2-7　集合划分问题。

问题描述：n 个元素的集合$\{1, 2, \cdots, n\}$可以划分为若干非空子集。例如，当 n=4 时，集合$\{1, 2, 3, 4\}$可以划分为 15 个不同的非空子集如下：

$\{\{1\}, \{2\}, \{3\}, \{4\}\}$　　$\{\{1, 3\}, \{2, 4\}\}$
$\{\{1, 2\}, \{3\}, \{4\}\}$　　$\{\{1, 4\}, \{2, 3\}\}$
$\{\{1, 3\}, \{2\}, \{4\}\}$　　$\{\{1, 2, 3\}, \{4\}\}$
$\{\{1, 4\}, \{2\}, \{3\}\}$　　$\{\{1, 2, 4\}, \{3\}\}$
$\{\{2, 3\}, \{1\}, \{4\}\}$　　$\{\{1, 3, 4\}, \{2\}\}$
$\{\{2, 4\}, \{1\}, \{3\}\}$　　$\{\{2, 3, 4\}, \{1\}\}$
$\{\{3, 4\}, \{1\}, \{2\}\}$　　$\{\{1, 2, 3, 4\}\}$
$\{\{1, 2\}, \{3, 4\}\}$

算法设计：给定正整数 n，计算出 n 个元素的集合$\{1, 2, \cdots, n\}$可以划分为多少个不同的非空子集。

数据输入：由文件 input.txt 提供输入数据。文件的第 1 行是元素个数 n。

结果输出：将计算出的不同的非空子集数输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	52

思路

使用斯特林数(Stirling numbers)和贝尔数(Bell numbers)可以有效地计算将 n 个元素的集合 $\{1,2,...,n\}$ 划分为非空子集的总数。

斯特林数 $S(n,k)$：将 n 个元素划分成 k 个非空子集的方式，存在这样的递推关系：

$$S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$$

(1)第一部分考虑的是将第 n 个元素加入到已经划分好的 k 个子集中。我们可以把新元素放入任意一个已经存在的 k 个子集中的 k 种选择。因此，这部分的结果是 $k \cdot S(n-1,k)$，表示将 $n-1$ 个元素划分为 k 个非空子集的方式，随后选择一个子集来放入新元素。

(2)第二部分考虑的是将第 n 个元素单独放入一个新的子集中。因此这部分的结果是 $S(n-1,k-1)$。

贝尔数 $B(n)$:表示将 n 个元素的集合划分为非空子集的总数，可以通过斯特林数计算：

$$B(n) = \sum_{k=1}^{n} S(n,k)$$

代码

```
// 实现题 2-7 集合划分问题
#include<bits/stdc++.h>
#define endl '\n'
typedef unsigned long long ull;
const int N = 10005;
using namespace std;
ull S[N][N];
int n;
void StirlingNumbers() {
    S[0][0] = 1; //s(0, 0) = 1
    for(int i = 1; i <= n; ++i) {
        for(int k = 1; k <= n; k++) {
            S[i][k] = k * S[i - 1][k] + S[i - 1][k - 1];
        }
    }
}
ull BellNumber(int n) {
    ull number = 0;
    for(int k = 1; k <= n; ++k) number += S[n][k]; //计算bell数
    return number;
}
int main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    ifstream ifs("testData/input3.txt");
    ofstream ofs("results/output3.txt");
    ifs >> n;
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    StirlingNumbers();
    ofs << BellNumber(n) << endl;
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    ofs << "time: " << duration.count() << " ms" << endl;
```

```
    ifs.close();
    ofs.close();
    return 0;
}
```

`关键代码及解释`：

```
void StirlingNumbers() {
    S[0][0] = 1; //S(0, 0) = 1
    for(int i = 1; i <= n; ++i) {
        for(int k = 1; k <= n; k++) {
            S[i][k] = k * S[i - 1][k] + S[i - 1][k - 1];
        }
    }
}
```

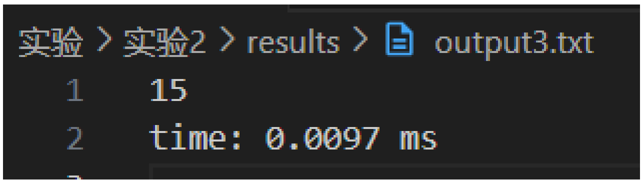
使用递推 $S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$ 求斯特林数。

```
ull BellNumber(int n) {
    ull number = 0;
    for(int k = 1; k <= n; ++k) number += S[n][k]; //计算bell数
    return number;
}
```

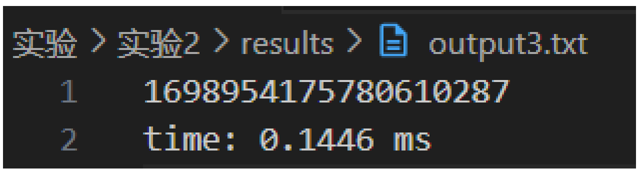
通过斯特林数求出贝尔数 $B(n) = \sum_{k=1}^{n} S(n,k)$，即划分的非空子集的总数。

测试结果

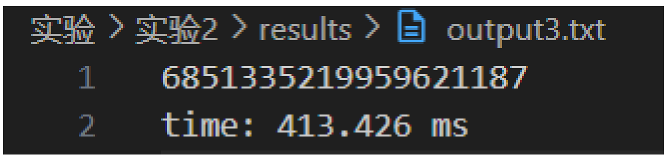
小规模：n=4



中规模：n=50



大规模：n=10000



算法分析

时间复杂度分析：计算斯特林数的嵌套循环中，外层循环迭代 n 次，内层循环迭代最多 n 次。因此，构建斯特林数表的时间复杂度为 $O(n^2)$。计算贝尔数时，循环遍历从 1 到 n，并且对每个 k 值从斯特林数表中获取值，时间复杂度为 $O(n)$。因此，时间复杂度为：

$$O(n^2)$$

空间复杂度分析：需要一个二维数组 S 构建斯特林数表，因此空间复杂度为：

$$O(n^2)$$