1 问题抽象

这个问题和 0/1 背包问题（ 0/1 Knapsack Problem ）类似，可以考虑借鉴其思路。先将问题抽象为如下的问题：

> 给定 $C > 0, w_i > 0, 1 \le i \le n$, 求一 0/1 序列 $(x_1, x_2, ..., x_n)$ 满足 $max \sum_{i=1}^{n} w_i x_i$, 要求 $\sum_{i=1}^{n} w_i x_i \le C, x_i \in 0, 1$

稍加思考可以发现，lab2 的问题和上面所描述的问题等价！

2 算法实现

2.1 算法思路

采用 dynamic programming(reference: slides.dynamic programming page29-33) 解决这一问题. 使用 dynamic programming 必须证明原问题具有优化子结构和重叠子问题. 以此保证问题解的正确性.

优化子结构 可以表示为如下的形式：

> 如果 $(x_1, x_2, ..., x_n)$ 是该问题的一个优化解，那么 $(x_2, ..., x_n)$ 是下面所描述的更小子问题的优化解：$max \sum_{i=2}^{n} w_i x_i$, 要求 $\sum_{i=2}^{n} w_i x_i \le C - w_1 x_1, x_i \in 0, 1$

Proof ：

反证法：如果 $(x_2, ..., x_n)$ 不是该子问题的优化解，则存在 $(y_2, ..., y_n)$ 是该子问题的优化解，使得 $(x_1, y_2, ..., y_n)$ 是原问题更优的解，与假设矛盾，得证！

重叠子问题

> 在较长序列 $(x_1, x_2, ..., x_n)$ 的求解过程中，可以看到，会多次求解较短序列的子问题，即该问题具有重叠子问题！

从而我们证明了原问题具有重叠子问题和优化子结构.

算法分析

设子问题 $max \sum_{k=i}^{n} w_k x_k, max \sum_{k=i}^{n} w_k x_k \le j, x_k \in 0, 1, i \le k \le n$ 的最优解代价为 $m[i, j]$, 则 $m[i, j]$ 是 *limit valuable* 为 j，可选物品为 $i, i+1, i+2, ..., n$ 时问题的最优解代价。

递归方程为：

$$m[i,j] = \begin{cases} m[i+1,j], 0 \ne j < w_i \\ \\ max(m[i+1,j], m[i+1,j-w_i] + w_i), j \ge w_i \end{cases}$$

边界条件为：

$$m[i,j] = \begin{cases} 0, 0 \ne j < w_n \\ \\ w_n, j \ge w_n \end{cases}$$

计算方法

1. 使用 $m[n,j] = 0$ 或 w_n 计算最底下一行
2. 使用递归方程计算其他 m[i,j]
3. 核心就是构造一个 $n \times w$ 阶矩阵 M，n 代表的是数的个数，w 代表的是每个数的大小，即是所谓的“权值”，通过构造一个备忘录矩阵来存储每一个子问题的最优解，例如 M_{ij} 就存储着 *limit value* 为 j，从 $i \to n$ 的最优解。

最优解构造问题

1. 设 $m[1,C]$ 是最优解代价值，构造最优解的过程如下：

```
    if(m[1][C] == m[2][C])
        x1=0;  //不选编号为1的元素
    else
        x1=1;  //选编号为1的元素
```

generate solution.cc

2. 如果 $x_1 = 0$，继续由 $m[2,C]$ 构造最优解
3. 如果 $x_1 = 1$，继续由 $m[2,C-x_1]$ 构造最优解

2.2 算法实现

编译环境：使用 UBUNTU 命令行界面编辑，编译器为 g++(Apple clang version 12.0.0 (clang-1200.0.32.27))，编译命令为：g++ ” /sort.cc” -o ” /sort.out” -W -Wall -O2 -std=c++17。硬件环境：Intel i9-9800H

具体代码如下：

一些 core code 的解释以注释的形式呈现在下面：

```
#include "bits/stdc++.h" //在本地目录下的库，oi打比赛的时候常用，这里就直接用了
using namespace std;


int maxNum(int a, int b) { return (a > b) ? a : b; }
//maxNum(a,b)返回两个int数中较大的一个

//insight(array,Matrix,weight array,number,limit value) 打印一个0/1数组array，长度为输入数据的
    大小number,处理的依据来自备忘录矩阵Matrix，保存着每一个子问题的优化解，weight array存储着
    每一个数据的大小（权值）
void insight(int *exist, int **val, int *w, int bagnum, int weight) {
  for (int i = bagnum; i > 0; i--) {
    if (val[i][weight] > val[i - 1][weight]) {
      exist[i] = 1;
      weight = weight - w[i];
    }
    //利用generate solution.cc进行递归
  }
  for (int j = 1; j <= bagnum; j++) cout << exist[j] << " ";
```

```
  //打印数组
}


//maxVal(Matrix, weight array, weight array, number, limit value)返回备忘录矩阵最大的值, 函数主要是在构造Matrix, 限制条件为limit value
int maxVal(int **val, int *v, int *w, int bagnum, int weight) {
  int i, j;
  for (i = 0; i <= bagnum; i++) val[i][0] = 0;
  for (i = 0; i <= weight; i++) val[0][i] = 0;
  // 用 **val 表示一个二维数组
  for (i = 1; i <= bagnum; i++) {
    for (j = 1; j <= weight; j++) {
      if (j < w[i])
        val[i][j] = val[i - 1][j];
      else
        val[i][j] = max(val[i - 1][j], val[i - 1][j - w[i]] + v[i]);
    }
    // 利用递归方程进行递归
  }
  return val[bagnum][weight];
  //返回右上角的值, 这个值就是原问题的优化解
}


// maxVal2和insight2的功能和前面的是一样的, 只不过返回类型是long long (对于我的laptop来说就是一个 64 比特的值, 这主要是为了在data2的测试当中防止指针溢出造成编译的segement fault)
long long maxVal2(long long **val, long long *v, long long *w, int bagnum,
                  long long weight) {
  int i, j;
  for (i = 0; i <= bagnum; i++) val[i][0] = 0;
  for (i = 0; i <= weight; i++) val[0][i] = 0;
  for (i = 1; i <= bagnum; i++) {
    for (j = 1; j <= weight; j++) {
      if (j < w[i])
        val[i][j] = val[i - 1][j];
      else
        val[i][j] = max(val[i - 1][j], val[i - 1][j - w[i]] + v[i]);
    }
  }
  return val[bagnum][weight];
}

void insight2(int *exist, long long **val, long long *w, int bagnum,
              long long weight) {
  for (int i = bagnum; i > 0; i--) {
    if (val[i][weight] > val[i - 1][weight]) {
      exist[i] = 1;
      weight = weight - w[i];
    }
  }
  for (int j = 1; j <= bagnum; j++) cout << exist[j] << " ";
}
```

```
69
70
71  //主函数
72  int main() {
73    int i, j;
74    int num;
75    cout << "which file do you want to check? print 1 if you want to check "
76            "data1.dat, print 2 if you want to check data2.dat "
77         << endl;
78    cin >> num;
79    ifstream fin;
80
81
82    switch (num) {
83      case 1: {
84        int bagnum, weight;
85        fin.open("/Users/edith_lzh/Desktop/c++/algorithm/lab2/data1.dat");
86        int l = 0;
87        int data[1001];
88        int arr[1001];
89        // int all[6];
90        for (l = 0; l < 2002; l++) {
91          if (l % 2 == 0) fin >> arr[l / 2];    // 1001 array, 1st is 0
92          if (l % 2 == 1) fin >> data[l / 2];  // 1001 array, 1st is limit
93        }
94        fin.close();
95        bagnum = arr[1000];  // numbers
96        weight = data[0];     // limit
97
98        int **val = new int *[bagnum + 1];  //这样定义二维数组, 是为了方便传入
99        val[0] = new int[(bagnum + 1) * (weight + 1)];
100       for (i = 1; i <= bagnum; i++) val[i] = val[i - 1] + weight + 1;
101
102       int *w = new int[bagnum + 1]();
103       for (i = 1; i <= bagnum; i++) w[i] = data[i];
104
105       int *v = new int[bagnum + 1]();
106       for (i = 1; i <= bagnum; i++) v[i] = w[i];
107
108       cout << "max sum is" << maxVal(val, v, w, bagnum, weight) << '\n';
109
110       int *exist = new int[bagnum]();  //为了查看我们选择了哪些数字
111       cout << "we get the bag num is:";
112       insight(exist, val, w, bagnum, weight);
113       cout << endl;
114       break;
115     }
116     case 2: {
117       int bagnum;
118       long long weight;
119       ifstream fin;
120       fin.open("/Users/edith_lzh/Desktop/c++/algorithm/lab2/data2.dat");
121       int l = 0;
```

```
        long long data[4];
        int arr[4];
        // int all[6];
        for (l = 0; l < 8; l++) {
          if (l % 2 == 0) fin >> arr[l / 2];    // 4 array, 1st is 0
          if (l % 2 == 1) fin >> data[l / 2];   // 4 array, 1st is limit
        }
        fin.close();

        bagnum = arr[3];    // numbers
        weight = data[0];   // limit

        long long **val =
            new long long *[bagnum + 1];   //这样定义二维数组，是为了方便传入
        val[0] = new long long[(bagnum + 1) * (weight + 1)];
        for (i = 1; i <= bagnum; i++) val[i] = val[i - 1] + weight + 1;

        long long *w = new long long[bagnum + 1]();
        for (i = 1; i <= bagnum; i++) w[i] = data[i];

        long long *v = new long long[bagnum + 1]();
        for (i = 1; i <= bagnum; i++) v[i] = w[i];

        cout << "max sum is" << maxVal2(val, v, w, bagnum, weight) << '\n';

        int *exist = new int[bagnum]();  //为了查看我们选择了哪些包
        cout << "we get the bag num is:";
        insight2(exist, val, w, bagnum, weight);
        cout << endl;
        break;
      }
    }
}
```

lab2.cc

2.3 程序性能分析及思考

1. 运行结果

data1.dat 的输出为 maxsum=1111111，选取数的编号为：1 2 3 5 7 8 12 13 14 15 16 18 19 20 21 23 24

```
which file do you want to check? print 1 if you want to check data1.dat, print 2 if you want to check data2.dat
2
max sum is1073741824
we get the bag num is:1 1 0
 edith_lzh@Edith  ~/Desktop/c++    main ±+
```

data2.dat 的输出为 maxsum=1073741824，选取数的编号为: 1 2

经过手工计算，算法输出的结果的确是正确答案.

2. *tips：我没有将数组规模作为参数传入 main 函数，而是直接读取文件，然后根据数据排布规律寻找确定位置上的值作为传入参数

3. 为了得到精确的运行时间, 我重新设计了代码, 将 cout 输出删去, 只考虑函数的运行时间, 用 c++11 chrono 库计时，计时代码和得到的运行时间结果如下所示:

code:

```cpp
auto start = std::chrono::steady_clock::now();
//code
auto end = std::chrono::steady_clock::now();
std::chrono::duration<double, std::milli> elapsed =
    end - start;  // std::milli 表示以毫秒为时间单位
std::cout << "time: " << elapsed.count() << "ms" << std::endl;

```

lab2.cc

运行结果截图:

```
which file do you want to check? print 1 if you want to check data1.dat, print 2 if you want to check data2.dat
1
time: 2423.98ms
 edith_lzh@Edith  ~/Desktop/c++    main ±+
```

data1.dat 的运行时间为 2423.98ms

```
which file do you want to check? print 1 if you want to check data1.dat, print 2 if you want to check data2.dat
2
time: 41509.5ms
 edith_lzh@Edith  ~/Desktop/c++    main ±+
```

data2.dat 的运行时间为 41509.5ms

4. 内存占用:
用系统自带的活动监视器，data1.dat 所需虚拟内存在 9G 左右，data2.dat 所需虚拟内存在 40G 左右。

5. 可以发现，dynamic programming 在处理数据量较大的小数据集时游刃有余，而在处理一些大数据集的时候捉襟见肘，内存占用和运行时间都非常不乐观。原因是明了的，因为在处理大数据集的时候，建立的备忘录矩阵也会扩展到相当大的尺度上，以 data2.dat 为例，建立的备忘录矩阵有多达 1073741824 列!，而对于矩阵中每一个子问题，又会建立一个相似大小的备忘录矩阵，这样的话，整个问题所需的内存会提升到一个巨大的量级！dynamic programming 不适合求解这样的问题。