**3-3.“漂亮打印”问题：**

（1）最优子结构性质：

假设在最优打印方案中，将第1~k个单词在第一行打印，则后面的第k+1~n个单词的打印方案也是子问题的最优解；

（2）重叠子问题：

在第一行打印第1~k个单词的所有方案，都要考虑后续单词的打印子问题，因此计算中有许多重叠的子问题；

（3）递归表达式：

在一行中打印第i~j个单词时，多余的空格数为：

当extra[i][j]<0时，说明一行打印不下全部单词，由于不允许将单词打破，故此时打印代价为；

当j=n时，说明已经打印到了最后一行，此时打印代价为0。

故将第i~j个单词打印到一行上的代价为：

假设c[j]表示将第1~j个单词“漂亮打印”的最小代价，则：

代码如下（时间复杂度与空间复杂度均为）：

#include <iostream>

#include <cmath>

#include <vector>

#include <algorithm>

#include <string>

using namespace std;

vector<string> article = { "Sun", "stressed", "that", "every", "resident", "in", "the", "municipality", "with", "a", "permanent", "population", "of", "25", "million", "must", "undergo", "nucleic", "acid", "tests", "in", "the", "citywide", "mass", "testing", ".", "She", "said", "it", "is", "necessary", "to", "expand", "makeshift", "hospitals", "and", "designated", "hospitals", "for", "COVID-19", "treatment", ",", "and", "required", "the", "preparation", "of", "sufficient", "quarantine", "venues", "and", "swift", "actions", "in", "transferring", "and", "treating", "patients", "."};

void beauty\_print(int\* L, int M, int n) {

//extra[i][j]表示将第i~j个单词打印到同一行时剩余的空格数

int\*\* extra = new int\* [n + 1];

for (int i = 0; i < n + 1; i++)

extra[i] = new int[n + 1];

//lc[i][j]表示将第i~j个单词打印到同一行时花费的代价

unsigned long\*\* lc = new unsigned long\* [n + 1];

for (int i = 0; i < n + 1; i++)

lc[i] = new unsigned long[n + 1];

//c[j]表示将第1~j个单词漂亮打印的最小代价

unsigned long\* c = new unsigned long[n + 1];

//s[j]表示将第1~j个单词漂亮打印时最后一行的开头单词

int\* s = new int[n + 1];

//初始化extra

for (int i = 1; i <= n; i++) {

for (int j = i; j <= n; j++) {

int sum = 0;

for (int k = i; k <= j; k++)

sum += L[k];

extra[i][j] = M - j + i - sum;

}

}

//初始化lc

for (int i = 1; i <= n; i++) {

for (int j = i; j <= n; j++) {

if (extra[i][j] < 0)

lc[i][j] = INT\_MAX;

else if (j == n)

lc[i][j] = 0;

else

lc[i][j] = (unsigned long)pow(extra[i][j], 3);

}

}

//求解子问题

c[0] = 0;

for (int j = 1; j <= n; j++) {

c[j] = lc[1][j];

for (int i = 2; i <= j; i++) {

unsigned long t = c[i - 1] + lc[i][j];

if (t < c[j]) {

c[j] = t;

s[j] = i;

}

}

}

//构造最优解

cout << "漂亮打印的最小代价：" << c[n] << endl;

for (int i = 0; i < M; i++)

cout << ' ';

cout << 'M' << endl;

vector<int> cut;

int k = s[n];

while (k > 0) {

cut.push\_back(k);

k = s[k - 1];

}

reverse(cut.begin(), cut.end());

int count = 0;

for (int i = 0; i < article.size(); i++) {

cout << article[i] << ' ';

if (count < cut.size() && i == cut[count] - 2) {

cout << endl;

count++;

}

}

//释放内存空间

for (int i = 0; i < n + 1; i++)

delete[] extra[i];

delete[] extra;

for (int i = 0; i < n + 1; i++)

delete[] lc[i];

delete[] lc;

delete[] c;

delete[] s;

}

//测试程序

int main(void) {

int M = 50;

int n = article.size();

int\* L = new int[n + 1];

for (int i = 1; i <= n; i++)

L[i] = article[i - 1].size();

beauty\_print(L, M, n);

delete[] L;

return 0;

}

**3-5.二维0-1背包问题：**

问题的形式化描述为：给定，，，，，，要求找出元0-1向量，，，使得达到最大，且满足下面的约束条件：

类比于一维的0-1背包问题，易知该问题同样具有最优子结构性质。

假设其子问题（背包容量为，容积为，可选择物品为）：

的最优值为，则：

原问题的最优解为。时间复杂度和空间复杂度均为。

代码如下：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

vector<int> weight = { 600, 400, 200, 200, 300 };

vector<int> volume = { 800, 400, 200, 200, 300 };

vector<int> value = { 8, 10, 4, 5, 5 };

void knapsack\_2d(const int &c, const int &d) {

const int n = value.size();

//分配内存空间，存储m(i,j,k)

int \*\*\*m = new int \*\*[n + 1];

for (int i = 0; i <= n; i++) {

m[i] = new int \*[c + 1];

for (int j = 0; j <= c; j++)

m[i][j] = new int[d + 1];

}

//分配内存空间，存储0-1向量

int \*x = new int[n + 1];

//初始化m(n,j,k)

for (int j = 0; j <= c; j++) {

for (int k = 0; k <= d; k++) {

m[n][j][k] = 0;

if (j >= weight[n - 1] && k >= volume[n - 1])

m[n][j][k] = value[n - 1];

}

}

//求解子问题

for (int i = n - 1; i > 0; i--) {

for (int j = 0; j <= c; j++) {

for (int k = 0; k <= d; k++) {

m[i][j][k] = m[i + 1][j][k];

if (j >= weight[i - 1] && k >= volume[i - 1])

m[i][j][k] = max(m[i + 1][j][k], m[i + 1][j - weight[i - 1]][k - volume[i - 1]] + value[i - 1]);

}

}

}

//构造最优解

int temp1 = c, temp2 = d;

for (int i = 1; i < n; i++) {

if (m[i][temp1][temp2] == m[i + 1][temp1][temp2])

x[i] = 0;

else {

x[i] = 1;

temp1 -= weight[i - 1];

temp2 -= volume[i - 1];

}

}

x[n] = (m[n][c][d] > 0) ? 1 : 0;

//输出

cout << "最大价值：" << m[1][c][d] << endl;

cout << "0-1向量：";

for (int i = 1; i <= n; i++)

cout << x[i] << ' ';

//释放内存空间

for (int i = 0; i <= n; i++) {

for (int j = 0; j <= c; j++)

delete[] m[i][j];

delete[] m[i];

}

delete[] m;

delete[] x;

}

//测试程序

int main(void) {

int c = 1000;

int d = 1000;

knapsack\_2d(c, d);

return 0;

}

**3-6.Ackerman函数：**

代码如下：

#include <iostream>

int Ackerman(const int& m, const int& n) {

if (m < 0 || n < 0)

return -1;

if (m == 0)

return n + 1;

int\* val = new int[m + 1];

int\* ind = new int[m + 1];

val[0] = 1;

ind[0] = 0;

for (int i = 1; i <= m; i++) {

val[i] = -1;

ind[i] = -2;

}

while (ind[m] < n) {

val[0]++;

ind[0]++;

for (int i = 0; i < m; i++) {

if (ind[i] == 1 && ind[i + 1] < 0) {

val[i + 1] = val[0];

ind[i + 1] = 0;

}

if (val[i + 1] == ind[i]) {

val[i + 1] = val[0];

ind[i + 1]++;

}

}

}

int ans = val[m];

delete[] val;

delete[] ind;

return ans;

}

//测试程序

int main(void) {

std::cout << Ackerman(3, 10);

return 0;

}