

算法报告

个人信息

- 数据科学与计算机学院
- 2018级 软工3班
- 18342075
- 米家龙

目录

算法报告

个人信息

目录

算法

Next Fit

First Fit

First Fit Decreasing

Best Fit

Best Fit Decreasing

模拟退火 搭配 FF(D) 和 BF(D)

Local Search 搭配 FF(D) 和 BF(D)

Taboo Search 搭配 FF(D) 和 BF(D)

测试结果

测试环境

测试结果表格

测试截图

Next Fit

First Fit

FFD

Best Fit

BFD

模拟退火

Local Search

Taboo Search

算法

Next Fit

思路：

- 始终只维持一个打开的箱子
- 对于每一个要装入的物品，检查该物品是否可以装入
 - 如果可以装入，则装入
 - 如果无法装入，则新开一个箱子，装入该物品

缺陷：每个物品只有放入当前箱子和空箱子的选择，因此面对 背包容量为10 数据为 2 9 3 8 4 7 10

1 这样的数据时，当最优解存在时，会出现最优解和结果近似相差一倍的箱子的情况，

$$NF(L) \leq 2 \cdot OPT(L) - 1$$

时间复杂度为 $O(|L|)$

代码实现如下，采用了**在线**的处理方式：

```
1  #include <iostream>
2  #include <time.h>
3  #include <vector>
4
5  using namespace std;
6
7  #define MAX_NUM 1000
8
9  time_t startTime, endTime; // 开始和结束时间
10
11 int goodsNum;           // 货物数量
12 int capacity;           // 箱子容量
13 vector<int> bins;       // 箱子
14 vector<int> goods;      // 货物队列
15 int bin;                // 当前箱子
16
17 /**
18  * 下项适应
19  */
20 void nextFit(int goodsNum);
21
22 int main()
23 {
24
25     cin >> goodsNum;
26     cin >> capacity;
27
28     bins.clear();
29     goods.clear();
30     bin = 0;
31
32     cout << "货物数量: " << goodsNum << '\t' << "背包容量: " << capacity << endl;
33
34     startTime = clock();
35
36     nextFit(goodsNum);
37
38     endTime = clock();
39
40     cout << "需要背包数量为: " << bins.size() << "\t 消耗时间: "
```

```

41         << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" << endl;
42
43     return 0;
44 }
45
46 void nextFit(int goodsNum)
47 {
48     for (int i = 0; i < goodsNum; i++)
49     {
50         int good = 0;
51         cin >> good;
52         if (good + bin <= capacity) // 如果能装下
53         {
54             bin += good;
55         }
56         else
57         {
58             bins.push_back(bin);
59             bin = good;
60         }
61         goods.push_back(good);
62         // cout << i << '\t' << good << endl;
63     }
64     bins.push_back(bin);
65 }

```

First Fit

思路：

- 维持所有的箱子打开
- 对于每一个要装入的物品，检查该物品是否可以装入打开的箱子
 - 如果可以装入，则装入
 - 如果无法装入，则新开一个箱子，装入该物品

缺陷：由于物品没有实现排序，则可能由于先装入小的物品，使大的物品在后来放入时无法装入，只得开启新的箱子，造成了空间的浪费

界限： $FF(L) \leq \lfloor 1.7OPT(L) \rfloor$

平均时间复杂度： $O(|L|\log(|L|))$

代码实现如下，采用了**在线**的处理方式：

```

1  #include <iostream>
2  #include <time.h>
3  #include <vector>
4
5  using namespace std;
6
7  #define MAX_NUM 1000
8
9  time_t startTime, endTime; // 开始和结束时间
10
11 int goodsNum;           // 货物数量
12 int capacity;           // 箱子容量
13 vector<int> bins;       // 箱子
14 vector<int> goods;      // 货物队列

```

```

15  int bin;           // 当前箱子
16
17  /**
18   * 首次适应
19   */
20  void firstFit(int goodsNum);
21
22  int main()
23  {
24      cin >> goodsNum;
25      cin >> capacity;
26
27      bins.clear();
28      goods.clear();
29      bin = 0;
30
31      cout << "货物数量: " << goodsNum << '\t' << "背包容量: " << capacity << endl;
32
33      startTime = clock();
34
35      firstFit(goodsNum);
36
37      endTime = clock();
38
39      cout << "需要背包数量为: " << bins.size() << "\t 消耗时间: "
40           << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" << endl;
41
42      return 0;
43  }
44
45  void firstFit(int goodsNum)
46  {
47      for (int i = 0; i < goodsNum; i++) // 在线输入
48      {
49          int good = 0;
50          bool findBin = false;
51          cin >> good;
52          for (int i = 0; i < bins.size(); i++)
53          {
54              if (bins[i] + good <= capacity) // 能够找到对应的箱子
55              {
56                  bins[i] += good;
57                  findBin = true;
58                  break;
59              }
60          }
61          if (!findBin) // 如果没有找到
62          {
63              bins.push_back(good);
64          }
65      }
66  }

```

First Fit Decreasing

降序首次适应算法，和 FF 算法相比，需要对物品先进行降序排序，再按照 FF 算法进行装箱

界限： $FDD(I) \leq \frac{11}{9}OPT(I) + \frac{2}{3}$

时空复杂度：随排序算法变动而变化

代码实现：

```
1  #include <iostream>
2  #include <time.h>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  #define MAX_NUM 1000
9
10 time_t startTime, endTime; // 开始和结束时间
11
12 int goodsNum;           // 货物数量
13 int capacity;           // 箱子容量
14 vector<int> bins;       // 箱子
15 vector<int> goods;      // 货物队列
16 int bin;                // 当前箱子
17
18 /**
19  * 首次适应
20  */
21 void firstFit(int goodsNum);
22
23 int main()
24 {
25     cin >> goodsNum;
26     cin >> capacity;
27
28     bins.clear();
29     goods.clear();
30     bin = 0;
31
32     cout << "货物数量: " << goodsNum << '\t' << "背包容量: " << capacity << endl;
33
34     startTime = clock();
35
36     firstFit(goodsNum);
37
38     endTime = clock();
39
40     cout << "需要背包数量为: " << bins.size() << "\t 消耗时间: "
41          << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" << endl;
42
43     return 0;
44 }
45
46 void firstFit(int goodsNum)
47 {
48     int good = 0;
49     for (int i = 0; i < goodsNum; i++) // 输入
```

```

50     {
51         cin >> good;
52         goods.push_back(good);
53     }
54
55     std::sort(goods.begin(), goods.end(), [](int a, int b) { return a > b; }); //
    倒序排序
56
57     for (int i = 0; i < goodsNum; i++)
58     {
59         bool findBin = false;
60         for (int j = 0; j < bins.size(); j++)
61         {
62             if (bins[j] + goods[i] <= capacity)
63             {
64                 bins[j] += goods[i];
65                 findBin = true;
66                 break;
67             }
68         }
69         if (!findBin)
70         {
71             bins.push_back(goods[i]);
72         }
73     }
74 }

```

Best Fit

思路：

- 维持所有的箱子打开
- 对于每一个要装入的物品，检查该物品是否可以装入打开的箱子
 - 如果可以装入，则需要判断该箱子是否是可装入箱子中剩余容量最小的
 - 如果是，则装入
 - 否，则跳过
 - 如果无法装入，则新开一个箱子，装入该物品

界限： $BF(L) \leq \lfloor 1.7OPT(L) \rfloor$

平均时间复杂度： $O(|L|\log(|L|))$

代码实现如下：

```

1  #include <iostream>
2  #include <time.h>
3  #include <vector>
4
5  using namespace std;
6
7  #define MAX_NUM 1000
8
9  time_t startTime, endTime; // 开始和结束时间
10
11 int goodsNum;           // 货物数量
12 int capacity;           // 箱子容量
13 vector<int> bins;       // 箱子

```

```

14  vector<int> goods; // 货物队列
15  int bin;           // 当前箱子
16
17  /**
18   * 最佳适应
19   */
20  void bestFit(int goodsNum);
21
22  int main()
23  {
24      cin >> goodsNum;
25      cin >> capacity;
26
27      bins.clear();
28      goods.clear();
29      bin = 0;
30
31      // cout << "货物数量: " << goodsNum << '\t' << "背包容量: " << capacity << endl;
32
33      startTime = clock();
34
35      bestFit(goodsNum);
36
37      endTime = clock();
38
39      cout << "需要背包数量为: " << bins.size() << "\t 消耗时间: "
40           << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" << endl;
41
42      return 0;
43  }
44
45  void bestFit(int goodsNum)
46  {
47      for (int i = 0; i < goodsNum; i++) // 在线输入
48      {
49          int good = 0;
50          int bestFitBin[] = {-1, 0}; // {序号, 已有重量}, 代表最适合的箱子
51          cin >> good;
52          for (int i = 0; i < bins.size(); i++)
53          {
54              if (bins[i] + good <= capacity) // 能够找到能放下的箱子
55              {
56                  if (bestFitBin[1] < bins[i]) // 比较剩余大小, 看是否是最适合的
57                  {
58                      bestFitBin[0] = i;
59                      bestFitBin[1] = bins[i];
60                  }
61              }
62          }
63          if (bestFitBin[0] == -1) // 如果没有找到
64          {
65              bins.push_back(good);
66          }
67          else
68          {
69              bins[bestFitBin[0]] += good;
70          }
71      }

```

Best Fit Decreasing

降序最佳适应算法，和 BF 算法相比，需要对物品先进行降序排序，再按照 BF 算法进行装箱

界限： $BFD(I) \leq \frac{11}{9}OPT(I) + \frac{2}{3}$

时空复杂度：随排序算法变动而变化

代码实现：

```

1  #include <iostream>
2  #include <time.h>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  #define MAX_NUM 1000
9
10 time_t startTime, endTime; // 开始和结束时间
11
12 int goodsNum;           // 货物数量
13 int capacity;           // 箱子容量
14 vector<int> bins;       // 箱子
15 vector<int> goods;      // 货物队列
16 int bin;                // 当前箱子
17
18 /**
19  * 最佳适应
20  */
21 void bestFit(int goodsNum);
22
23 int main()
24 {
25     cin >> goodsNum;
26     cin >> capacity;
27
28     bins.clear();
29     goods.clear();
30     bin = 0;
31
32     startTime = clock();
33
34     bestFit(goodsNum);
35
36     endTime = clock();
37
38     cout << "需要背包数量为: " << bins.size() << "\t 消耗时间: "
39          << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" << endl;
40
41     return 0;
42 }
43
44 void bestFit(int goodsNum)
45 {
46     int good = 0;
47     for (int i = 0; i < goodsNum; i++) // 输入

```



```

48     {
49         cin >> good;
50         goods.push_back(good);
51     }
52
53     std::sort(goods.begin(), goods.end(), [](int a, int b) { return a > b; }); //
    倒序排序
54
55     for (int j = 0; j < goods.size(); j++)
56     {
57         int bestFitBin[] = {-1, 0}; // {序号, 已有重量}, 代表最适合的箱子
58         for (int i = 0; i < bins.size(); i++)
59         {
60             if (bins[i] + goods[j] <= capacity) // 能够找到能放下的箱子
61             {
62                 if (bestFitBin[1] < bins[i]) // 比较剩余大小, 看是否是最适合的
63                 {
64                     bestFitBin[0] = i;
65                     bestFitBin[1] = bins[i];
66                 }
67             }
68         }
69         if (bestFitBin[0] == -1) // 如果没有找到
70         {
71             bins.push_back(goods[j]);
72         }
73         else
74         {
75             bins[bestFitBin[0]] += goods[j];
76         }
77     }
78 }

```

模拟退火 搭配 FF(D) 和 BF(D)

- 先使用 FFD 和 BFD 获取第一次的结果
- 进行降温迭代
- 每随机生成一个新的序列, 便通过 FF 和 BF 分别计算当前情况下的解
 - 如果当前解更优, 则替换当前序列和解
 - 否则仍然有几率获取到较差的解

代码实现如下:

```

1  #include <iostream>
2  #include <math.h>
3  #include <vector>
4  #include <algorithm>
5
6  using namespace std;
7
8  int goodsNum;
9  int capacity;
10
11 #define MAX_LOOP_TIME 200
12 #define RAND_TIME 100
13

```

```

14 double k = 0.1;
15 double r = 0.97;    //用于控制降温的快慢
16 double T = 300;    //系统的温度，系统初始应该要处于一个高温的状态
17 double T_min = 0.1; //温度的下限，若温度T达到T_min，则停止搜索
18                      //返回指定范围内的随机浮点数
19
20 double dEFF, dEBF, current;
21
22 vector<int> goods;
23 vector<int> bins;
24 vector<int> newGoods; // 用于储存新生成的序列
25
26 int bestResult; // 最好的结果
27
28 /**
29  * 获取货物列表
30  */
31 void getGoods();
32
33 /**
34  * 产生 (dbLow, dbUpper) 之间的随机数
35  * @param dbLow double 下限
36  * @param dbUpper double 上限
37  */
38 double rnd(double dbLow, double dbUpper);
39
40 /**
41  * 模拟退火
42  */
43 int simulatedFire();
44
45 /**
46  * 获取新的货物顺序
47  */
48 void getOneNewGoodsList();
49
50 /**
51  * 首次适应算法（不能排序）
52  */
53 int firstFit();
54
55 /**
56  * 最佳适应算法（不能排序）
57  */
58 int bestFit();
59
60 /**
61  * 用新的序列覆盖原来的序列
62  */
63 void copyGoods();
64
65 int main()
66 {
67     time_t startTime, endTime; // 开始与结束时间
68
69     cin >> goodsNum >> capacity;
70
71     goods.clear();

```

```

72     bins.clear();
73     newGoods.clear();
74
75     startTime = clock();
76     srand((unsigned)(time(NULL))); // 初始化随机种子，避免生成同样的随机结果
77
78     getGoods();
79     simulatedFire();
80
81     endTime = clock();
82
83     cout << "需要背包数量为: " << bestResult << "\t 消耗时间: "
84           << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" <<
endl;
85
86     return 0;
87 }
88
89 void getGoods()
90 {
91     int good = 0;
92     for (int i = 0; i < goodsNum; i++)
93     {
94         cin >> good;
95         goods.push_back(good);
96         newGoods.push_back(good);
97     }
98
99     sort(goods.begin(), goods.end(), [](int a, int b) { return a > b; });
100    sort(newGoods.begin(), newGoods.end(), [](int a, int b) { return a > b; });
101 }
102
103 double rnd(double dbLow, double dbUpper)
104 {
105     double dbTemp = rand() / ((double)RAND_MAX + 1.0);
106     return dbLow + dbTemp * (dbUpper - dbLow);
107 }
108
109 int simulatedFire()
110 {
111     int a = firstFit();
112     int b = bestFit();
113     bestResult = a > b ? b : a;
114     while (T > T_min)
115     {
116         for (int i = 0; i < MAX_LOOP_TIME; i++)
117         {
118             getOneNewGoodsList(); // 随机生成新的序列
119             int ffBins = firstFit();
120             int bfBins = bestFit();
121
122             dEFF = ffBins - bestResult;
123             dEBF = bfBins - bestResult;
124
125             if (dEFF < 0 || dEBF < 0) // 如果有更优解
126             {
127                 bestResult = ffBins < bfBins ? ffBins : bfBins;
128                 copyGoods();

```

```

129     }
130     else // 一定概率接受较差解
131     {
132         if (exp(-dEFF / (T * k)) > rnd(0, 1))
133         {
134             copyGoods();
135         }
136         else if (exp(-dEBF / (T * k)) > rnd(0, 1))
137         {
138             copyGoods();
139         }
140     }
141
142     // cout << ffBins << '\t' << bfBins << '\t' << bestResult << endl;
143 }
144 T = r * T; // 降温退火
145 }
146 }
147
148 void getOneNewGoodsList()
149 {
150     newGoods.clear();
151     for (auto i = goods.begin(); i != goods.end(); i++)
152     {
153         newGoods.push_back(*i);
154     }
155
156     // 生成随机序列
157     for (int i = 0; i < RAND_TIME; i++)
158     {
159         int tmp;
160
161         // 生成交换的位置
162         int a = rand() % goodsNum;
163         int b = rand() % goodsNum;
164
165         tmp = newGoods[a];
166         newGoods[a] = newGoods[b];
167         newGoods[b] = tmp;
168     }
169 }
170
171 void copyGoods()
172 {
173     goods.clear();
174     for (int i = 0; i < newGoods.size(); i++)
175     {
176         goods.push_back(newGoods[i]);
177     }
178 }
179
180 int firstFit()
181 {
182     bins.clear();
183     for (int i = 0; i < newGoods.size(); i++)
184     {
185         bool findBin = false;
186         for (int j = 0; j < bins.size(); j++)

```

```

187     {
188         if (bins[j] + newGoods[i] <= capacity) // 找到了能放下的就放进去
189         {
190             bins[j] += newGoods[i];
191             findBin = true;
192             break;
193         }
194     }
195     if (!findBin) // 否则开一个新的箱子
196     {
197         bins.push_back(newGoods[i]);
198     }
199 }
200
201 return bins.size();
202 }
203
204 int bestFit()
205 {
206     bins.clear();
207     for (int i = 0; i < newGoods.size(); i++)
208     {
209         int findBestBin[] = {-1, 0};
210         for (int j = 0; j < bins.size(); j++)
211         {
212             // 如果找到了能放下的, 并且剩余容量更小
213             if (bins[j] + newGoods[i] <= capacity && findBestBin[1] < bins[j])
214             {
215                 findBestBin[0] = j;
216                 findBestBin[1] = bins[j];
217             }
218         }
219         if (findBestBin[0] == -1)
220         {
221             bins.push_back(newGoods[i]);
222         }
223         else
224         {
225             bins[findBestBin[0]] += newGoods[i];
226         }
227     }
228
229     return bins.size();
230 }

```

Local Search 搭配 FF(D) 和 BF(D)

每随机生成一个新序列, 使用 FF 和 BF 判断是否是更优解, 当没有出现更优解次数达到阈值时, 退出迭代

该算法缺陷是容易陷入局部最优解, 具体代码实现:

```

1  #include <iostream>
2  #include <math.h>
3  #include <vector>
4  #include <algorithm>
5

```

```

6   using namespace std;
7
8   int goodsNum;
9   int capacity;
10
11  #define MAX_LOOP_TIME 300
12  #define RAND_TIME 100
13
14  vector<int> goods;
15  vector<int> bins;
16  vector<int> newGoods; // 用于储存新生成的序列
17
18  int bestResult;          // 最好的结果
19  int noBetterResultTime = 0; // 连续没有出现更优解的次数
20
21  /**
22   * 获取货物列表
23   */
24  void getGoods();
25
26  /**
27   * LS 算法
28   */
29  int localSearch();
30
31  /**
32   * 获取新的货物顺序
33   */
34  void getOneNewGoodsList();
35
36  /**
37   * 首次适应算法（不能排序）
38   */
39  int firstFit();
40
41  /**
42   * 最佳适应算法（不能排序）
43   */
44  int bestFit();
45
46  /**
47   * 用新的序列覆盖原来的序列
48   */
49  void copyGoods();
50
51  int main()
52  {
53      time_t startTime, endTime; // 开始与结束时间
54
55      cin >> goodsNum >> capacity;
56
57      goods.clear();
58      bins.clear();
59      newGoods.clear();
60
61      startTime = clock();
62      srand((unsigned)(time(NULL))); // 初始化随机种子，避免生成同样的随机结果
63
64      getGoods();

```

```

64     localSearch();
65
66     endTime = clock();
67
68     cout << "需要背包数量为: " << bestResult << "\t 消耗时间: "
69         << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" <<
endl;
70
71     return 0;
72 }
73
74 void getGoods()
75 {
76     int good = 0;
77     for (int i = 0; i < goodsNum; i++)
78     {
79         cin >> good;
80         goods.push_back(good);
81         newGoods.push_back(good);
82     }
83
84     sort(goods.begin(), goods.end(), [](int a, int b) { return a > b; });
85     sort(newGoods.begin(), newGoods.end(), [](int a, int b) { return a > b; });
86 }
87
88 int localSearch()
89 {
90     int a = firstFit();
91     int b = bestFit();
92     bestResult = a > b ? b : a;
93     while (noBetterResultTime < MAX_LOOP_TIME)
94     {
95         getOneNewGoodsList(); // 随机生成新的序列
96         int ffBins = firstFit();
97         int bfBins = bestFit();
98
99         if (bestResult > ffBins || bestResult > bfBins) // 如果有更优解
100         {
101             bestResult = ffBins < bfBins ? ffBins : bfBins;
102             copyGoods();
103             noBetterResultTime = 0;
104         }
105         else
106         {
107             noBetterResultTime++;
108         }
109     }
110 }
111
112 void getOneNewGoodsList()
113 {
114     newGoods.clear();
115     for (auto i = goods.begin(); i != goods.end(); i++)
116     {
117         newGoods.push_back(*i);
118     }
119
120     // 生成随机序列

```

```

121     for (int i = 0; i < RAND_TIME; i++)
122     {
123         int tmp;
124
125         // 生成交换的位置
126         int a = rand() % goodsNum;
127         int b = rand() % goodsNum;
128
129         tmp = newGoods[a];
130         newGoods[a] = newGoods[b];
131         newGoods[b] = tmp;
132     }
133 }
134
135 void copyGoods()
136 {
137     goods.clear();
138     for (int i = 0; i < newGoods.size(); i++)
139     {
140         goods.push_back(newGoods[i]);
141     }
142 }
143
144 int firstFit()
145 {
146     bins.clear();
147     for (int i = 0; i < newGoods.size(); i++)
148     {
149         bool findBin = false;
150         for (int j = 0; j < bins.size(); j++)
151         {
152             if (bins[j] + newGoods[i] <= capacity) // 找到了能放下的就放进去
153             {
154                 bins[j] += newGoods[i];
155                 findBin = true;
156                 break;
157             }
158         }
159         if (!findBin) // 否则开一个新的箱子
160         {
161             bins.push_back(newGoods[i]);
162         }
163     }
164
165     return bins.size();
166 }
167
168 int bestFit()
169 {
170     bins.clear();
171     for (int i = 0; i < newGoods.size(); i++)
172     {
173         int findBestBin[] = {-1, 0};
174         for (int j = 0; j < bins.size(); j++)
175         {
176             // 如果找到了能放下的, 并且剩余容量更小
177             if (bins[j] + newGoods[i] <= capacity && findBestBin[1] < bins[j])
178             {

```



```

179         findBestBin[0] = j;
180         findBestBin[1] = bins[j];
181     }
182 }
183 if (findBestBin[0] == -1)
184 {
185     bins.push_back(newGoods[i]);
186 }
187 else
188 {
189     bins[findBestBin[0]] += newGoods[i];
190 }
191 }
192
193 return bins.size();
194 }

```

Taboo Search 搭配 FF(D) 和 BF(D)

局部搜索的局限在于达到局部的时候便会结束，而禁忌搜索则抛弃掉这个局限，只是用迭代次数作为硬性要求，并且通过禁忌表来储存已经出现的结果，从而有更多的可能获得最优解

具体思路：

1. 在迭代次数未结束时，做以下循环操作：
 1. 以元数据的降序排列为基准，使用 BF 和 FF 得到最初的结果
 2. 随机进行数据交换，生成一组新的序列，判断这个序列是否在禁忌表中
 - 如果在，则是的该禁忌对象长度 + 1
 - 如果不在，则将生产一个禁忌对象，并将其加入到禁忌列表中，基础长度为 $\sqrt{\text{货物序列长度}}$
 3. 计算出对应的 FF 和 BF 结果，比较储存的局部最优解
 - 如果有则替换掉储存的局部最优解
 - 否则跳过
 4. 迭代次数 - 1
2. 返回结果

具体实现：

```

1  #include <iostream>
2  #include <vector>
3  #include <time.h>
4  #include <math.h>
5  #include <algorithm>
6
7  using namespace std;
8
9  #define MAX_ITERATION_TIME 1000 // 最大迭代次数
10 #define RAND_TIME 100 // 随机次数
11
12 vector<int> goods;
13 vector<int> newGoods;
14 vector<int> bins;
15
16 int goodsNum; // 货物数量
17 int capacity; // 容量

```

```
18  int bestResult; // 最佳结果
19
20  // 禁忌对象
21  struct TabooObject
22  {
23      int length;          // 禁忌长度（生命周期）
24      int bins;            // 需要箱子数量（其实好像用不到）
25      vector<int> goodsList; // 储存货物顺序的向量
26  };
27
28  int bestInTabooList = 0; // 这个没用上
29
30  vector<struct TabooObject> tabooList; // 禁忌列表
31
32  /**
33   * 获取输入
34   */
35  void getGoods();
36
37  /**
38   * 复制 vector<int>
39   * @param from - vector<int> 被复制的向量
40   * @param &to - vector<int> 目的向量
41   */
42  void copyGoods(vector<int> from, vector<int> &to);
43
44  /**
45   * 首次适应算法
46   * @return 返回需要的箱子
47   */
48  int firstFit();
49
50  /**
51   * 最佳适应算法
52   * @return 返回需要的箱子
53   */
54  int bestFit();
55
56  /**
57   * 主要迭代步骤
58   */
59  void iterate();
60
61  /**
62   * 所有禁忌对象长度 - 1
63   * 如果有变为0的，就删除掉
64   */
65  void deleteTabooList();
66
67  /**
68   * 在禁忌列表中寻找是否有对应的向量
69   */
70  int findInTabooList(vector<int> list);
71
72  int main()
73  {
74      time_t startTime, endTime;
75      cin >> goodsNum >> capacity;
```

```

76     srand((unsigned)(time(NULL)));
77
78     startTime = clock();
79     getGoods();
80     copyGoods(goods, newGoods);
81     int ffBins = firstFit();
82     int bfBins = bestFit();
83     bestResult = ffBins > bfBins ? bfBins : ffBins;
84     iterate();
85
86     for (int i = 0; i < MAX_ITERATION_TIME; i++)
87     {
88         iterate();
89     }
90
91     endTime = clock();
92
93     cout << "需要背包数量为: " << bestResult
94          << "\t 消耗时间: "
95          << (double)(endTime - startTime) / CLOCKS_PER_SEC * 1000 << " ms" <<
endl;
96 }
97
98 void getGoods()
99 {
100     int good;
101     for (int i = 0; i < goodsNum; i++)
102     {
103         cin >> good;
104         goods.push_back(good);
105     }
106
107     sort(goods.begin(), goods.end(), [](int a, int b) { return a > b; });
108 }
109
110 void copyGoods(vector<int> from, vector<int> &to)
111 {
112     to.clear();
113     for (auto i = from.begin(); i != from.end(); i++)
114     {
115         to.push_back(*i);
116     }
117 }
118
119 void getNewGoodsList()
120 {
121     copyGoods(goods, newGoods);
122     for (int i = 0; i < RAND_TIME; i++)
123     {
124         int a = rand() % goodsNum;
125         int b = rand() % goodsNum;
126
127         int tmp = newGoods[a];
128         newGoods[a] = newGoods[b];
129         newGoods[b] = tmp;
130     }
131 }
132

```

```

133 void iterate()
134 {
135     getNewGoodsList();
136     int ffBins = firstFit();
137     int bfBins = bestFit();
138
139     // 如果没找到, 就需要加入到禁忌表中
140     if (findInTabooList(newGoods) == -1)
141     {
142         struct TabooObject tmp;
143         tmp.bins = ffBins < bfBins ? ffBins : bfBins;
144         // tmp.bins = sqrt(MAX_ITERATION_TIME);
145         tmp.bins = sqrt(goodsNum);
146         copyGoods(newGoods, tmp.goodsList);
147         tabooList.push_back(tmp);
148     }
149
150     // 如果是更优解, 就更新
151     if (bestResult > ffBins || bestResult > bfBins)
152     {
153         bestResult = ffBins < bfBins ? ffBins : bfBins;
154         copyGoods(newGoods, goods);
155     }
156
157     // 更新禁忌表
158     deleteTabooList();
159 }
160
161 int findInTabooList(vector<int> list)
162 {
163     for (int i = 0; i < tabooList.size(); i++) // 遍历禁忌表
164     {
165         bool isSame = true;
166         // 遍历当前禁忌对象, 判断是否和传入向量等价
167         for (int j = 0; j < tabooList[i].goodsList.size(); j++)
168         {
169             if (tabooList[i].goodsList[j] != list[j])
170             {
171                 isSame = false;
172                 break;
173             }
174         }
175         if (isSame) // 如果等价, 则禁忌长度加1, 返回该对象的索引
176         {
177             tabooList[i].length++;
178             return i;
179         }
180     }
181     // 否则返回 -1
182     return -1;
183 }
184
185 int firstFit()
186 {
187     bins.clear();
188     for (int i = 0; i < newGoods.size(); i++)
189     {
190         bool findBin = false;

```

```

191     for (int j = 0; j < bins.size(); j++)
192     {
193         if (bins[j] + newGoods[i] <= capacity) // 找到了能放下的就放进去
194         {
195             bins[j] += newGoods[i];
196             findBin = true;
197             break;
198         }
199     }
200     if (!findBin) // 否则开一个新的箱子
201     {
202         bins.push_back(newGoods[i]);
203     }
204 }
205
206 return bins.size();
207 }
208
209 int bestFit()
210 {
211     bins.clear();
212     for (int i = 0; i < newGoods.size(); i++)
213     {
214         int findBestBin[] = {-1, 0};
215         for (int j = 0; j < bins.size(); j++)
216         {
217             // 如果找到了能放下的, 并且剩余容量更小
218             if (bins[j] + newGoods[i] <= capacity && findBestBin[1] < bins[j])
219             {
220                 findBestBin[0] = j;
221                 findBestBin[1] = bins[j];
222             }
223         }
224         if (findBestBin[0] == -1)
225         {
226             bins.push_back(newGoods[i]);
227         }
228         else
229         {
230             bins[findBestBin[0]] += newGoods[i];
231         }
232     }
233
234     return bins.size();
235 }
236
237 void deleteTabooList()
238 {
239     for (auto i = tabooList.begin(); i != tabooList.end(); i++)
240     {
241         i->length--;
242     }
243
244     sort(tabooList.begin(), tabooList.end(), [](struct TabooObject a, struct
245     TabooObject b) { return a.length > b.length; }); // 排序, 好删掉是0的
246
247     for (int i = tabooList.size() - 1; i >= 0; i--)
248     {

```

```
248     if (tabooList[i].length == 0)
249     {
250         tabooList.pop_back();
251     }
252     else // 不为0直接中断
253     {
254         break;
255     }
256 }
257 }
```

测试结果

测试环境

代码运行环境为 WSL:

```
1  Linux LAPTOP-QTCGESHO 4.4.0-19041-Microsoft #1-Microsoft Fri Dec 06 14:06:00 PST
    2019 x86_64 x86_64 x86_64 GNU/Linux
```

使用脚本进行批处理输入输出:

```
1  #!/bin/bash
2  # ScriptName: bbp.sh
3  cppFile=$@;          # 获取参数
4  DataPath="./Data";   # 测试文件路径
5  compiler="g++";       # 编译器
6  txt=$(ls $DataPath); # 获取测试文件列表
7
8  # 如果输入为空,就提示
9  if [ ! -n "$cppFile" ]; then
10     echo "请输入代码文件";
11     exit 0;
12 fi
13
14 # echo $cppFile;
15 ${compiler} $cppFile;
16
17 for file in $txt
18 do
19     echo "测试文件为 $file";
20     "./a.out" < $DataPath/$file;
21     echo ;
22 done
23
24 rm "./a.out";
```

测试结果表格

测试文件\时间 ms 背包数量	Next Fit	First Fit	Best Fit
Waescher_TEST0005.txt	0 33	0 29	0 29
Waescher_TEST0014.txt	0 28	0 24	0 24
Waescher_TEST0022.txt	0 16	0 15	0 15
Waescher_TEST0030.txt	0 31	0 28	0 28
Waescher_TEST0044.txt	0 15	0 15	0 15
Waescher_TEST0049.txt	0 12	0 12	0 12
Waescher_TEST0054.txt	0 15	0 15	0 15
Waescher_TEST0055A.txt	0 17	0 16	0 16
Waescher_TEST0055B.txt	0 22	0 21	0 21
Waescher_TEST0058.txt	0 23	0 21	0 21
Waescher_TEST0065.txt	0 18	0 16	0 16
Waescher_TEST0068.txt	0 13	0 13	0 13
Waescher_TEST0075.txt	0 14	0 14	0 14
Waescher_TEST0082.txt	0 33	0 25	0 25
Waescher_TEST0084.txt	0 18	0 17	0 17
Waescher_TEST0095.txt	0 18	0 17	0 17
Waescher_TEST0097.txt	0 13	0 13	0 13

由于测试数据已经经过排序，因此在测试结果上 FFD 和 BFD 与 FF 和 BF 没有区别

测试文件\ (时间 ms 背包数量)	First Fit Decreasing	Best Fit Decreasing
Waescher_TEST0005.txt	0 29	0 29
Waescher_TEST0014.txt	0 24	0 24
Waescher_TEST0022.txt	0 15	0 15
Waescher_TEST0030.txt	0 28	0 28
Waescher_TEST0044.txt	0 15	0 15
Waescher_TEST0049.txt	0 12	0 12
Waescher_TEST0054.txt	0 15	0 15
Waescher_TEST0055A.txt	0 16	0 16
Waescher_TEST0055B.txt	0 21	0 21
Waescher_TEST0058.txt	0 21	0 21
Waescher_TEST0065.txt	0 16	0 16
Waescher_TEST0068.txt	0 13	0 13
Waescher_TEST0075.txt	0 14	0 14
Waescher_TEST0082.txt	0 25	0 25
Waescher_TEST0084.txt	0 17	0 17
Waescher_TEST0095.txt	0 17	0 17
Waescher_TEST0097.txt	0 13	0 13

测试文件\时间 ms 背包数量	Simulated Fire	Local Search	Taboo Search
Waescher_TEST0005.txt	1593.75 29	15.625 29	5,056.5 29
Waescher_TEST0014.txt	1,296.88 24	0 24	4,781.25 24
Waescher_TEST0022.txt	640.625 15	15.625 15	4,640.62 15
Waescher_TEST0030.txt	1,500 28	0 28	4,968.75 28
Waescher_TEST0044.txt	1,406.25 15	15.625 15	5,000 15
Waescher_TEST0049.txt	1,109.38 11 ~ 12	0 12	4,937.5 11 ~ 12
Waescher_TEST0054.txt	1,296.88 15	15.625 15	4,968.75 15
Waescher_TEST0055A.txt	1,296.88 16	15.625 16	4,953.12 16
Waescher_TEST0055B.txt	2,406.25 21	15.625 21	5,203.12 21
Waescher_TEST0058.txt	1,046.88 21	15.625 21	4,812.5 21
Waescher_TEST0065.txt	640.625 16	0 16	4,640.62 16
Waescher_TEST0068.txt	1,281.25 13	0 13	4,968.75 13
Waescher_TEST0075.txt	1,796.88 14	0 14	5,078.12 14
Waescher_TEST0082.txt	1,140.62 25	0 25	4,718.25 25
Waescher_TEST0084.txt	937.5 17	0 17	4,750 17
Waescher_TEST0095.txt	1,421.88 17	0 17	4,953.12 17
Waescher_TEST0097.txt	1,046.88 13	0 13	4,859.38 13

在测试模拟退火后，发现其运行速度变慢，但是最终结果没有特别优化，因此怀疑是数据问题，因此自己生成了一个长度为1000的测试数据进行额外的测试，测试结果如下

算法	时间(ms)	结果
NF	0	624
FF	0	486
BF	0	482
FFD	0	470
BFD	0	470
SF	100,000 ~ 132,156	470
LS	900 ~ 1,000	470
TS	11,109.4	470

测试截图

Next Fit

```
需要背包数量为: 23      消耗时间: 0 ms  
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./next_fit.cpp
```

```
测试文件为 Waescher_TEST0005.txt  
货物数量: 114   背包容量: 10000  
需要背包数量为: 33      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0014.txt  
货物数量: 96    背包容量: 10000  
需要背包数量为: 28      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0022.txt  
货物数量: 57    背包容量: 10000  
需要背包数量为: 16      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0030.txt  
货物数量: 111   背包容量: 10000  
需要背包数量为: 31      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0044.txt  
货物数量: 164   背包容量: 10000  
需要背包数量为: 15      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0049.txt  
货物数量: 141   背包容量: 10000  
需要背包数量为: 12      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0054.txt  
货物数量: 144   背包容量: 10000  
需要背包数量为: 15      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0055A.txt  
货物数量: 142   背包容量: 10000  
需要背包数量为: 17      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0055B.txt  
货物数量: 239   背包容量: 10000  
需要背包数量为: 22      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0058.txt  
货物数量: 91    背包容量: 10000  
需要背包数量为: 23      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0065.txt  
货物数量: 60    背包容量: 10000  
需要背包数量为: 18      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0068.txt  
货物数量: 163   背包容量: 10000  
需要背包数量为: 13      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0075.txt  
货物数量: 228   背包容量: 10000  
需要背包数量为: 14      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0082.txt  
货物数量: 86    背包容量: 10000  
需要背包数量为: 33      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0084.txt  
货物数量: 92    背包容量: 10000  
需要背包数量为: 18      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0095.txt  
货物数量: 153   背包容量: 10000  
需要背包数量为: 18      消耗时间: 0 ms
```

```
测试文件为 Waescher_TEST0097.txt  
货物数量: 119   背包容量: 10000  
需要背包数量为: 13      消耗时间: 0 ms
```

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ █
```



First Fit

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./first_fit.cpp
```

测试文件为 Waescher_TEST0005.txt

货物数量: 114 背包容量: 10000

需要背包数量为: 29 消耗时间: 0 ms

测试文件为 Waescher_TEST0014.txt

货物数量: 96 背包容量: 10000

需要背包数量为: 24 消耗时间: 0 ms

测试文件为 Waescher_TEST0022.txt

货物数量: 57 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0030.txt

货物数量: 111 背包容量: 10000

需要背包数量为: 28 消耗时间: 0 ms

测试文件为 Waescher_TEST0044.txt

货物数量: 164 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0049.txt

货物数量: 141 背包容量: 10000

需要背包数量为: 12 消耗时间: 0 ms

测试文件为 Waescher_TEST0054.txt

货物数量: 144 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0055A.txt

货物数量: 142 背包容量: 10000

需要背包数量为: 16 消耗时间: 0 ms

测试文件为 Waescher_TEST0055B.txt

货物数量: 239 背包容量: 10000

需要背包数量为: 21 消耗时间: 0 ms

测试文件为 Waescher_TEST0058.txt

货物数量: 91 背包容量: 10000

需要背包数量为: 21 消耗时间: 0 ms

测试文件为 Waescher_TEST0065.txt

货物数量: 60 背包容量: 10000

需要背包数量为: 16 消耗时间: 0 ms

测试文件为 Waescher_TEST0068.txt

货物数量: 163 背包容量: 10000

需要背包数量为: 13 消耗时间: 0 ms

测试文件为 Waescher_TEST0075.txt

货物数量: 228 背包容量: 10000

需要背包数量为: 14 消耗时间: 0 ms

测试文件为 Waescher_TEST0082.txt

货物数量: 86 背包容量: 10000

需要背包数量为: 25 消耗时间: 0 ms

测试文件为 Waescher_TEST0084.txt

货物数量: 92 背包容量: 10000

需要背包数量为: 17 消耗时间: 0 ms

测试文件为 Waescher_TEST0095.txt

货物数量: 153 背包容量: 10000

需要背包数量为: 17 消耗时间: 0 ms

测试文件为 Waescher_TEST0097.txt

货物数量: 119 背包容量: 10000

需要背包数量为: 13 消耗时间: 0 ms

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ █
```



FFD

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./first_fit_decreasing.cpp
```

测试文件为 Waescher_TEST0005.txt

货物数量: 114 背包容量: 10000

需要背包数量为: 29 消耗时间: 0 ms

测试文件为 Waescher_TEST0014.txt

货物数量: 96 背包容量: 10000

需要背包数量为: 24 消耗时间: 0 ms

测试文件为 Waescher_TEST0022.txt

货物数量: 57 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0030.txt

货物数量: 111 背包容量: 10000

需要背包数量为: 28 消耗时间: 0 ms

测试文件为 Waescher_TEST0044.txt

货物数量: 164 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0049.txt

货物数量: 141 背包容量: 10000

需要背包数量为: 12 消耗时间: 0 ms

测试文件为 Waescher_TEST0054.txt

货物数量: 144 背包容量: 10000

需要背包数量为: 15 消耗时间: 0 ms

测试文件为 Waescher_TEST0055A.txt

货物数量: 142 背包容量: 10000

需要背包数量为: 16 消耗时间: 0 ms

测试文件为 Waescher_TEST0055B.txt

货物数量: 239 背包容量: 10000

需要背包数量为: 21 消耗时间: 0 ms

测试文件为 Waescher_TEST0058.txt

货物数量: 91 背包容量: 10000

需要背包数量为: 21 消耗时间: 0 ms

测试文件为 Waescher_TEST0065.txt

货物数量: 60 背包容量: 10000

需要背包数量为: 16 消耗时间: 0 ms

测试文件为 Waescher_TEST0068.txt

货物数量: 163 背包容量: 10000

需要背包数量为: 13 消耗时间: 0 ms

测试文件为 Waescher_TEST0075.txt

货物数量: 228 背包容量: 10000

需要背包数量为: 14 消耗时间: 0 ms

测试文件为 Waescher_TEST0082.txt

货物数量: 86 背包容量: 10000

需要背包数量为: 25 消耗时间: 0 ms

测试文件为 Waescher_TEST0084.txt

货物数量: 92 背包容量: 10000

需要背包数量为: 17 消耗时间: 0 ms

测试文件为 Waescher_TEST0095.txt

货物数量: 153 背包容量: 10000

需要背包数量为: 17 消耗时间: 0 ms

测试文件为 Waescher_TEST0097.txt

货物数量: 119 背包容量: 10000

需要背包数量为: 13 消耗时间: 0 ms

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ █
```

Best Fit

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh best_fit.cpp
测试文件为 Waescher_TEST0005.txt
需要背包数量为：29      消耗时间：0 ms

测试文件为 Waescher_TEST0014.txt
需要背包数量为：24      消耗时间：0 ms

测试文件为 Waescher_TEST0022.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0030.txt
需要背包数量为：28      消耗时间：0 ms

测试文件为 Waescher_TEST0044.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0049.txt
需要背包数量为：12      消耗时间：0 ms

测试文件为 Waescher_TEST0054.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0055A.txt
需要背包数量为：16      消耗时间：0 ms

测试文件为 Waescher_TEST0055B.txt
需要背包数量为：21      消耗时间：0 ms

测试文件为 Waescher_TEST0058.txt
需要背包数量为：21      消耗时间：0 ms

测试文件为 Waescher_TEST0065.txt
需要背包数量为：16      消耗时间：0 ms

测试文件为 Waescher_TEST0068.txt
需要背包数量为：13      消耗时间：0 ms

测试文件为 Waescher_TEST0075.txt
需要背包数量为：14      消耗时间：0 ms

测试文件为 Waescher_TEST0082.txt
需要背包数量为：25      消耗时间：0 ms

测试文件为 Waescher_TEST0084.txt
需要背包数量为：17      消耗时间：0 ms

测试文件为 Waescher_TEST0095.txt
需要背包数量为：17      消耗时间：0 ms

测试文件为 Waescher_TEST0097.txt
需要背包数量为：13      消耗时间：0 ms
```


BFD

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./best_fit_decreasing.cpp
测试文件为 Waescher_TEST0005.txt
需要背包数量为：29      消耗时间：0 ms

测试文件为 Waescher_TEST0014.txt
需要背包数量为：24      消耗时间：0 ms

测试文件为 Waescher_TEST0022.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0030.txt
需要背包数量为：28      消耗时间：0 ms

测试文件为 Waescher_TEST0044.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0049.txt
需要背包数量为：12      消耗时间：0 ms

测试文件为 Waescher_TEST0054.txt
需要背包数量为：15      消耗时间：0 ms

测试文件为 Waescher_TEST0055A.txt
需要背包数量为：16      消耗时间：0 ms

测试文件为 Waescher_TEST0055B.txt
需要背包数量为：21      消耗时间：0 ms

测试文件为 Waescher_TEST0058.txt
需要背包数量为：21      消耗时间：0 ms

测试文件为 Waescher_TEST0065.txt
需要背包数量为：16      消耗时间：0 ms

测试文件为 Waescher_TEST0068.txt
需要背包数量为：13      消耗时间：0 ms

测试文件为 Waescher_TEST0075.txt
需要背包数量为：14      消耗时间：0 ms

测试文件为 Waescher_TEST0082.txt
需要背包数量为：25      消耗时间：0 ms

测试文件为 Waescher_TEST0084.txt
需要背包数量为：17      消耗时间：0 ms

测试文件为 Waescher_TEST0095.txt
需要背包数量为：17      消耗时间：0 ms

测试文件为 Waescher_TEST0097.txt
需要背包数量为：13      消耗时间：0 ms
```

模拟退火

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./simulated_fire.cpp
测试文件为 Waescher_TEST0005.txt
需要背包数量为: 29      消耗时间: 1593.75 ms

测试文件为 Waescher_TEST0014.txt
需要背包数量为: 24      消耗时间: 1296.88 ms

测试文件为 Waescher_TEST0022.txt
需要背包数量为: 15      消耗时间: 640.625 ms

测试文件为 Waescher_TEST0030.txt
需要背包数量为: 28      消耗时间: 1500 ms

测试文件为 Waescher_TEST0044.txt
需要背包数量为: 15      消耗时间: 1406.25 ms

测试文件为 Waescher_TEST0049.txt
需要背包数量为: 12      消耗时间: 1109.38 ms

测试文件为 Waescher_TEST0054.txt
需要背包数量为: 15      消耗时间: 1296.88 ms

测试文件为 Waescher_TEST0055A.txt
需要背包数量为: 16      消耗时间: 1296.88 ms

测试文件为 Waescher_TEST0055B.txt
需要背包数量为: 21      消耗时间: 2406.25 ms

测试文件为 Waescher_TEST0058.txt
需要背包数量为: 21      消耗时间: 1046.88 ms

测试文件为 Waescher_TEST0065.txt
需要背包数量为: 16      消耗时间: 640.625 ms

测试文件为 Waescher_TEST0068.txt
需要背包数量为: 13      消耗时间: 1281.25 ms

测试文件为 Waescher_TEST0075.txt
需要背包数量为: 14      消耗时间: 1796.88 ms

测试文件为 Waescher_TEST0082.txt
需要背包数量为: 25      消耗时间: 1140.62 ms

测试文件为 Waescher_TEST0084.txt
需要背包数量为: 17      消耗时间: 937.5 ms

测试文件为 Waescher_TEST0095.txt
需要背包数量为: 17      消耗时间: 1421.88 ms

测试文件为 Waescher_TEST0097.txt
需要背包数量为: 13      消耗时间: 1046.88 ms
```

Local Search

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh local_search.cpp
```

```
测试文件为 Waescher_TEST0005.txt  
需要背包数量为：29      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0014.txt  
需要背包数量为：24      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0022.txt  
需要背包数量为：15      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0030.txt  
需要背包数量为：28      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0044.txt  
需要背包数量为：15      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0049.txt  
需要背包数量为：12      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0054.txt  
需要背包数量为：15      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0055A.txt  
需要背包数量为：16      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0055B.txt  
需要背包数量为：21      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0058.txt  
需要背包数量为：21      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0065.txt  
需要背包数量为：16      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0068.txt  
需要背包数量为：13      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0075.txt  
需要背包数量为：14      消耗时间：15.625 ms
```

```
测试文件为 Waescher_TEST0082.txt  
需要背包数量为：25      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0084.txt  
需要背包数量为：17      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0095.txt  
需要背包数量为：17      消耗时间：0 ms
```

```
测试文件为 Waescher_TEST0097.txt  
需要背包数量为：13      消耗时间：0 ms
```

Taboo Search

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$ ./bbp.sh ./taboo_search.cpp
```

```
测试文件为 Waescher_TEST0005.txt
```

```
需要背包数量为：29      消耗时间：5062.5 ms
```

```
测试文件为 Waescher_TEST0014.txt
```

```
需要背包数量为：24      消耗时间：4781.25 ms
```

```
测试文件为 Waescher_TEST0022.txt
```

```
需要背包数量为：15      消耗时间：4640.62 ms
```

```
测试文件为 Waescher_TEST0030.txt
```

```
需要背包数量为：28      消耗时间：4968.75 ms
```

```
测试文件为 Waescher_TEST0044.txt
```

```
需要背包数量为：15      消耗时间：5000 ms
```

```
测试文件为 Waescher_TEST0049.txt
```

```
需要背包数量为：12      消耗时间：4937.5 ms
```

```
测试文件为 Waescher_TEST0054.txt
```

```
需要背包数量为：15      消耗时间：4968.75 ms
```

```
测试文件为 Waescher_TEST0055A.txt
```

```
需要背包数量为：16      消耗时间：4953.12 ms
```

```
测试文件为 Waescher_TEST0055B.txt
```

```
需要背包数量为：21      消耗时间：5203.12 ms
```

```
测试文件为 Waescher_TEST0058.txt
```

```
需要背包数量为：21      消耗时间：4812.5 ms
```

```
测试文件为 Waescher_TEST0065.txt
```

```
需要背包数量为：16      消耗时间：4640.62 ms
```

```
测试文件为 Waescher_TEST0068.txt
```

```
需要背包数量为：13      消耗时间：4968.75 ms
```

```
测试文件为 Waescher_TEST0075.txt
```

```
需要背包数量为：14      消耗时间：5078.12 ms
```

```
测试文件为 Waescher_TEST0082.txt
```

```
需要背包数量为：25      消耗时间：4718.75 ms
```

```
测试文件为 Waescher_TEST0084.txt
```

```
需要背包数量为：17      消耗时间：4750 ms
```

```
测试文件为 Waescher_TEST0095.txt
```

```
需要背包数量为：17      消耗时间：4953.12 ms
```

```
测试文件为 Waescher_TEST0097.txt
```

```
需要背包数量为：13      消耗时间：4859.38 ms
```

```
mijialong@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法/Project$
```