杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

程序设计实践报告

基于 C 语言的背包问题回溯法解题报告

卓越学院
计算机科学英才班
周靖凯

目录

1 引言

背包问题作为计算机科学中的经典问题，长期以来吸引了众多研究者的关注。它不仅在理论计算领域具有重要地位，而且在实际应用中极具价值，如在资源分配、财务规划等多个领域均有广泛应用。传统的背包问题侧重于在重量或体积限制下的价值最大化，而现实世界中的问题往往更为复杂，涉及多个限制维度，如重量、体积及物品件数等。

多维度背包问题增加了额外的限制条件，如物品的件数限制，使得问题更加复杂。这种增加的复杂性不仅挑战了传统算法的有效性，也促使研究者探索更为高效的解决方案。

本文介绍了一种解决增加了物品件数限制的多维度背包问题的算法。我们采用了深度优先搜索（DFS）的方法，通过递归探索所有可能的物品组合，寻找在给定重量、体积和件数限制下的最优解。本研究的主要贡献在于提出了一种简单直观的方法来处理这一复杂问题，并通过剪枝策略优化了算法的效率。

本文首先介绍了算法的核心思路和实现细节，随后进行了性能分析，探讨了算法在不同情景下的表现和潜在的优化空间。最后，我们对算法的实际应用价值和未来的改进方向进行了讨论。

2 问题描述

本问题是经典背包问题的一个变种，考虑物品的重量、体积和价值三个维度。给定 n 种物品，每种物品的重量为 w_i，体积为 v_i，价值为 p_i。背包的最大装重量为 W，最大体积为 V，物品不能超过件数 C。目标是选择一组物品放入背包，使得装入的物品总价值最大化，同时不超过背包的重量和体积限制。

3 算法思路

3.1 深度优先搜索（DFS）

算法的核心是使用深度优先搜索（DFS）策略。这种方法通过递归地探索所有可能的物品组合，以寻找最优解。在每一步，算法决定是否将当前物品加入背包，并相应地更新当前的总重量、总体积和总价值。

3.2 递归函数结构

递归函数 `dfs` 接受当前考虑的物品集合、背包的重量和体积限制、当前总价值、剩余可加入的物品件数以及当前考虑的物品索引。函数在考虑完所有物品后返回当前总价值。对于每件物品，算法考虑不加入该物品的情况以及在满足重量、体积和件数限制的情况下加入该物品的情况，并返回这两种选择中总价值较大的一种。

3.3 物品件数限制

与传统背包问题相比，本算法在每次递归调用时减少剩余可加入物品的件数，以确保在达到最大件数限制时停止添加新物品。

3.4 算法实现

在 `main` 函数中，算法首先读取物品的数量、背包的重量和体积限制以及最大可携带物品件数。然后，为每件物品分配空间并读取其重量、体积和价值。最后，调用 `knapsack` 函数，该函数进一步调用 `dfs` 函数计算并输出背包中物品的最大总价值。

4 性能分析

4.1 深度优先搜索（DFS）的复杂度

本算法的核心是深度优先搜索（DFS）。在最坏情况下，DFS 的时间复杂度为 $O(2^n)$，其中 n 是物品的数量。这种复杂度源于对每件物品加入或不加入背包的决定。然而，在实际应用中，由于背包的重量、体积限制或物品件数限制，搜索可能会提前终止，因此实际性能可能优于理论最大复杂度。

4.2 递归调用的影响

算法使用递归来实现 DFS，递归调用的深度可能对性能产生显著影响。在物品数量较多的情况下，递归深度的增加可能导致栈溢出或性能下降。优化递归调用或考虑迭代方法可能有助于提高性能。

4.3 剪枝优化

算法中的剪枝步骤有效减少了搜索空间。当当前组合的重量或体积超过限制，或者可加入的物品件数为零时，算法将停止进一步搜索。这种策略有助于提高算法的整体效率。

4.4 内存使用

由于每次递归调用都可能产生新的变量实例，算法的内存使用随递归深度的增加而增加。在处理大规模数据时，可能导致显著的内存使用。优化数据结构和减少变量复制可以降低内存消耗。

4.5 综合评估

总体来说，尽管该算法理论上可以解决多维度背包问题，但在处理大量物品时可能面临性能和内存使用方面的挑战。通过实施剪枝策略和优化内存使用，可以在一定程度上提升实际性能。对于大规模问题，可能需要更高效的算法，如动态规划或贪心算法。

5 小结

本文介绍的算法旨在解决增加了物品件数限制的多维度背包问题。通过实现深度优先搜索（DFS）策略，算法能够探索所有可能的物品组合，以寻找在满足重量、体积和件数限制条件下的最大价值组合。

5.1 算法的优势

算法的主要优势在于其简洁性和直观性。通过递归实现，算法能够清晰地表达问题的决策过程。此外，算法中的剪枝步骤有助于提高效率，特别是在部分组合明显不符合条件时，能够快速排除。

5.2 潜在的改进

尽管当前实现在小到中等规模的数据集上表现良好，但在处理大规模数据集时，性能和内存使用仍有改进空间。未来的工作可以集中在优化递归调用、减少内存消耗，或者探索更高效的算法，如动态规划。

A 附录

A.1 完整测试代码

knapsack.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int w, v, p;
} Goods;

int dfs(const Goods a[], const int n, const int retw,
   const int retv, const int sump, const int retc, const
   int cur)
{
    if (cur >= n)
        return sump;
    int res = dfs(a, n, retw, retv, sump, retc, cur + 1);
    if (retw >= a[cur].w && retv >= a[cur].v && retc >= 1)
    {
        int now = dfs(a, n, retw - a[cur].w, retv - a[cur
           ].v, sump + a[cur].p, retc - 1, cur + 1);
        if (now > res)
            res = now;
    }
    return res;
}

int knapsack(const Goods a[], const int n, const int w,
   const int v, const int c)
```

```
{
    return dfs(a, n, w, v, 0, c, 0);
}

int main()
{
    int n, w, v, c;
    scanf("%d%d%d%d", &n, &w, &v, &c);
    Goods *a = (Goods *)malloc(sizeof(Goods) * n);
    for (int i = 0; i < n; i++)
        scanf("%d%d%d", &a[i].w, &a[i].v, &a[i].p);
    printf("%d", knapsack(a, n, w, v, c));
    free(a);
    return 0;
}
```

杭州電子科技大学
HANGZHOU DIANZI UNIVERSITY

程序设计实践报告

基于 C 语言的非递归快速排序解题报告

卓越学院
计算机科学英才班
周靖凯

目录

1 引言

快速排序算法，由 C. A. R. Hoare 于 1960 年提出，是计算机科学中最高效和广泛使用的排序算法之一。以其优异的平均时间复杂度——$O(n \log n)$，它在处理大规模数据集时表现出色。然而，传统的快速排序算法通常通过递归实现，这在处理极大数据集时可能导致栈溢出的问题。

为此，本报告采用了非递归方法实现快速排序，有效避免了递归实现中的栈溢出风险。非递归实现通过使用栈结构来模拟递归调用的过程，既保持了算法的核心效率，又提高了其在处理大数据集时的稳定性。接下来，本文将详细介绍非递归快速排序算法的设计、实现，以及性能分析。

2 基本原理与实现

2.1 过程

快速排序的工作原理是通过分治的方式来将一个数组排序。

快速排序分为三个过程：

1. 将数列划分为两部分（要求保证相对大小关系）；
2. 递归到两个子序列中分别进行快速排序；
3. 不用合并，因为此时数列已经完全有序。

和归并排序不同，第一步并不是直接分成前后两个序列，而是在分的过程中要保证相对大小关系。具体来说，第一步要是要把数列分成两个部分，然后保证前一个子数列中的数都小于后一个子数列中的数。为了保证平均时间复杂度，一般是随机选择一个数 m 来当做两个子数列的分界。

之后，维护一前一后两个指针 p 和 q，依次考虑当前的数是否放在了应该放的位置（前还是后）。如果当前的数没放对，比如说如果后面的指针 q 遇到了一个比 m 小的数，那么可以交换 p 和 q 位置上的数，再把 p 向后移一位。当前的数的位置全放对后，再移动指针继续处理，直到两个指针相遇。

其实，快速排序没有指定应如何具体实现第一步，不论是选择 m 的过程还是划分的过程，都有不止一种实现方法。

第三步中的序列已经分别有序且第一个序列中的数都小于第二个数，所以直接拼接起来就好了。

2.2 稳定性

快速排序是一种不稳定的排序算法。

2.3 时间复杂度

快速排序的最优时间复杂度和平均时间复杂度为 $O(n \log n)$，最坏时间复杂度为 $O(n^2)$。

2.4 三路快速排序

三路快速排序（英语：3-way Radix Quicksort）是快速排序和基数排序的混合。它的算法思想基于荷兰国旗问题的解法。

与原始的快速排序不同，三路快速排序在随机选取分界点 m 后，将待排数列划分为三个部分：小于 m、等于 m 以及大于 m。这样做即实现了将与分界元素相等的元素聚集在分界元素周围这一效果。

2.5 递归实现

```
void quick_sort(int a[], const int n)
{
    if (n <= 1)
        return;
    int pivot = a[rand() % n];
    int i = 0, j = 0, k = n;
    while (i < k)
    {
        if (a[i] < pivot)
            swap(a[i++], a[j++]);
        else if (pivot < a[i])
            swap(a[i], a[--k]);
        else
            i++;
    }
    quick_sort(a, j);
    quick_sort(a + k, n - k);
    return;
}
```

2.6 非递归实现

尽管传统的快速排序通常采用递归来实现，但递归方法在处理极大数据集时可能遇到栈溢出的问题。为解决这一问题，非递归实现采用了栈来模拟递归过程，从而避免了递归带来的栈溢出风险。非递归快速排序在逻辑上保持了与递归版本相同的分区过程，但在执行方式上，它通过显式地使用栈来管理分区步骤的序列，从而实现了算法的迭代版本。

```
typedef struct
{
    int start, end;
} Range;

void quick_sort_stack(int a[], const int n)
{
    if (n <= 0)
        return;
    Range *r = (Range *)malloc(sizeof(Range) * n);
    int p = 0;
    r[p++] = (Range){0, n - 1};
    while (p)
    {
        Range range = r[--p];
        int len = range.end - range.start + 1;
        int pivot = a[rand() % len + range.start];
        int i = range.start, j = range.start, k = range.
           end;
        while (i <= k)
        {
            if (a[i] < pivot)
            {
                int t = a[i];
                a[i] = a[j];
```

```
                a[j] = t;
                i++, j++;
            }
            else if (pivot < a[i])
            {
                int t = a[i];
                a[i] = a[k];
                a[k] = t;
                k--;
            }
            else
                i++;
        }
        if (range.start < j - 1)
            r[p++] = (Range){range.start, j - 1};
        if (k + 1 < range.end)
            r[p++] = (Range){k + 1, range.end};
    }
    free(r);
    return;
}
```

3 程序设计

3.1 数据结构

程序中实现了一个简单但高效的栈结构，用于替代递归调用。栈由若干个节点组成，每个节点包含一个范围（`struct Range`），表示待排序数组的一部分。这个范围结构由两个整型字段组成：`start` 和 `end`，它们分别标记了数组中需要排序的段的起始和结束位置。

3.2 核心函数

程序的核心是 `quick_sort` 函数，它使用上述栈来实现快速排序的非递归版本。首先，函数将整个数组的范围压入栈中。然后，在栈不为空的情况下，它连续弹出范围并对其进行排序。排序过程中，选取范围内的随机元素作为枢纽，并基于这个枢纽将范围内的元素分为小于、等于和大于枢纽的三部分。每次分区后，将产生的新范围压回栈中，直到栈为空，表明排序完成。

4 算法实现

4.1 分区策略

算法的核心在于其分区操作，这是快速排序的关键步骤。在每次处理栈中的一个范围时，算法首先选择一个随机枢纽（`pivot`）。这一策略有助于减少对输入数据分布的依赖，从而在平均情况下保持良好的性能。随后，算法将范围内的元素根据它们与枢纽的比较结果进行重新排列：小于枢纽的元素移至其左侧，大于枢纽的元素则移至右侧。

4.2 循环控制

在传统的递归快速排序中，每一次分区都伴随着新的递归调用。而在非递归实现中，这一过程通过显式的循环控制来实现。使用栈结构存储每个分区操作后产生的新范围。当栈不为空时，算法持续弹出栈顶元素（即当前处理的范围），执行分区操作，然后根据分区结果将新的范围压入栈中。这种方法确保了算法能够处理整个数据集，同时避免了递归调用的栈溢出风险。

5 测试与结果

5.1 测试方法

测试分为两部分：正确性验证和性能评估。正确性验证通过将算法的输出与标准排序函数的输出进行比较来进行。这包括了对多种不同大小和特性的数据集（如随机数据、几乎升序数据和几乎逆序数据）的测试。性能评估则通过测量算法在不同大小的数据集上的运行时间来进行，特别关注其在大数据集上的表现。

5.2 测试结果

5.2.1 无序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.005587	0.060396	0.697338
非递归实现	0.005690	0.059575	0.665750

5.2.2 几乎升序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.003163	0.029129	0.337156
非递归实现	0.003154	0.030876	0.317150

5.2.3 几乎降序序列

运行时间（秒）	$n = 10^5$	$n = 10^6$	$n = 10^7$
递归实现	0.003090	0.030066	0.319865
非递归实现	0.003022	0.028127	0.320204

测试结果表明，非递归快速排序算法在所有测试数据集上均正确地完成了排序任务。在性能方面，算法显示出与递归版本相似的时间效率，特别是在大数据集上，其表现优于或等同于传统的递归实现。这些结果验证了非递归快速排序作为一种有效且可靠的排序方法的实用性。

5.3 性能分析

5.3.1 时间复杂度

在平均情况下，快速排序的时间复杂度为 $O(n \log n)$，这归因于每次分区操作大约将数据集分为两等分，并递归地应用于这些子集。非递归版本维持了这一性能特点，尽管实现方式由递归转变为使用栈进行迭代。对于普通快速排序，在最坏情况下，当数据已经是有序的或接近有序时，时间复杂度会达到 $O(n^2)$。然而，通过随机选择枢纽，算法的平均性能得以优化，这一策略有助于减少对输入数据分布的依赖。

5.3.2 空间复杂度

非递归快速排序的主要空间开销来自于栈的使用。在最坏的情况下，栈的大小可能需要与输入数组的大小成比例，导致空间复杂度达到 $O(n)$。然而，在平均情况下，由于分区操作的效率，栈的大小通常远小于数组的大小，使得实际的空间开销保持在较低水平。

5.3.3 优化可能性

尽管非递归快速排序已经非常高效，但仍有优化空间。例如，改进枢纽选择算法以进一步减少对特定数据分布的依赖，或采用更复杂的数据结构来降低在最坏情况下的空间需求。此外，对于小范围的数据集，可以考虑使用插入排序或其他更适合小数据集的排序算法来优化性能。

6　小结

本报告详细介绍了非递归快速排序算法的设计、实现和性能分析。通过深入探讨，我们可以得出以下几点结论：

非递归实现的快速排序算法在保持了传统快速排序高效性能的同时，成功避免了递归实现可能导致的栈溢出问题。这一点在处理大规模数据集时尤为重要，增强了算法的实用性和稳定性。

非递归实现在空间复杂度方面可能略高于递归版本，但它提供了更高的可靠性，尤其是在系统栈资源有限的环境中。此外，通过采用随机枢纽选择策略，算法成功地减少了对输入数据分布的敏感性，从而提高了其在各种数据集上的平均性能。

综上所述，非递归快速排序算法不仅是一种高效的排序方法，而且是一种适应性强、稳定可靠的解决方案，适用于各种规模的数据排序需求。

A 附录

A.1 完整测试代码

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

typedef struct
{
    int start, end;
} Range;

void quick_sort_stack(int a[], const int n)
{
    if (n <= 0)
        return;
    Range *r = (Range *)malloc(sizeof(Range) * n);
    int p = 0;
    r[p++] = (Range){0, n - 1};
    while (p)
    {
        Range range = r[--p];
        int len = range.end - range.start + 1;
        int pivot = a[rand() % len + range.start];
        int i = range.start, j = range.start, k = range.
           end;
        while (i <= k)
        {
            if (a[i] < pivot)
```

```
            {
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
                i++,j++;
            }
            else if (pivot < a[i])
            {
                int t=a[i];
                a[i]=a[k];
                a[k]=t;
                k--;
            }
            else
                i++;
        }
        if (range.start < j - 1)
            r[p++] = (Range){range.start, j - 1};
        if (k + 1 < range.end)
            r[p++] = (Range){k + 1, range.end};
    }
    free(r);
    return;
}

void quick_sort(int a[], const int n)
{
    if (n <= 1)
        return;
    int pivot = a[rand() % n];
    int i = 0, j = 0, k = n;
```

```
58      while (i < k)
59      {
60          if (a[i] < pivot)
61          {
62              int t = a[i];
63              a[i] = a[j];
64              a[j] = t;
65              i++, j++;
66          }
67          else if (pivot < a[i])
68          {
69              k--;
70              int t = a[i];
71              a[i] = a[k];
72              a[k] = t;
73          }
74          else
75              i++;
76      }
77      quick_sort(a, j);
78      quick_sort(a + k, n - k);
79      return;
80  }
81
82  bool is_sorted(int a[], int n)
83  {
84      for (int i = 1; i < n; i++)
85          if (a[i] < a[i - 1])
86              return false;
87      return true;
88  }
```

```
89  int cmp(const void *a, const void *b)
90  {
91      return (*(int *)a - *(int *)b);
92  }
93
94  void generate(int *a, int *b, int *c, int n)
95  {
96      for (int i = 0; i < n; i++)
97          a[i] = b[i] = rand();
98      qsort(b, n, sizeof(int), cmp);
99      for (int i = 0; i < n; i++)
100         c[i] = b[n - i - 1];
101     int times = 10;
102     for (int i = 1; i <= times; i++)
103     {
104         int l = rand() % n, r = rand() % n;
105         while (l == r)
106             l = rand() % n, r = rand() % n;
107         int t = b[l];
108         b[l] = b[r];
109         b[r] = t;
110     }
111     for (int i = 1; i <= times; i++)
112     {
113         int l = rand() % n, r = rand() % n;
114         while (l == r)
115             l = rand() % n, r = rand() % n;
116         int t = c[l];
117         c[l] = c[r];
118         c[r] = t;
119     }
```

```
120     return;
121 }
122 
123 void test(int a[], int n, void (*sort)(int *, int), const
       char type[], const char name[])
124 {
125     int *c = (int *)malloc(sizeof(int) * n);
126     for (int i = 0; i < n; i++)
127         c[i] = a[i];
128     int st = clock();
129     sort(c, n);
130     int ed = clock();
131     double time = (double)(ed - st) / CLOCKS_PER_SEC;
132     printf("%20s | %25s | time: %.6lf\n", type, name, time
           );
133     if (!is_sorted(c, n))
134     {
135         fprintf(stderr, "Error\n");
136         exit(1);
137     }
138     free(c);
139     return;
140 }
141 
142 int main(int argc, char *argv[])
143 {
144     if (argc < 3)
145     {
146         fprintf(stderr, "Need more parameters");
147         return 1;
148     }
```

```
149     int n = atoi(argv[1]), seed = atoi(argv[2]);
150     printf("n = %d, seed = %d\n", n, seed);
151     srand(seed);
152     int *a = (int *)malloc(sizeof(int) * n), *b = (int *)
            malloc(sizeof(int) * n), *c = (int *)malloc(sizeof(
            int) * n);
153     generate(a, b, c, n);
154     test(a, n, quick_sort, "Quick Sort", "Random Sequence"
            );
155     test(b, n, quick_sort, "Quick Sort", "Almost Ascending
             Sequence");
156     test(c, n, quick_sort, "Quick Sort", "Almost
            Descending Sequence");
157     test(a, n, quick_sort_stack, "Quick Sort Stack", "
            Random Sequence");
158     test(b, n, quick_sort_stack, "Quick Sort Stack", "
            Almost Ascending Sequence");
159     test(c, n, quick_sort_stack, "Quick Sort Stack", "
            Almost Descending Sequence");
160     return 0;
161 }
```