西安电子科技大学
XIDIAN UNIVERSITY

算法分析与设计

实验报告

实验名称：经典算法综合实验

任课教师：李青洋老师

学号姓名：

一、 分治（综合性实验）

(一) 实验题目

实验 1-S-1:排序算法大比拼。实现至少 3 种基于分治策略的排序算法,可选择以递归或/和循环方式实现其中部分或全部算法,测试时要求能够验证不同求解算法在不同数据场景下计算效率各有优势的特性,鼓励检索新实际工程数据集进行验证(要求标明所采用的实际数据集名称和对应的 URL)

(二) 实验原理

实现三种基于分治的排序（升序）算法：归并排序，快速排序（三分），堆排序:

归并排序:

分：把数组对半分开——选取中间元素做分割点，对左右子数组做递归

治+合并：创建以左右子数组赋值的临时数组，都在末尾增加一个哨兵（无穷大的元素）。然后逐个比较两临时数组中的元素，小的一个被放入原数组。

哨兵是为了在其中一个子数组元素被全部放入原数组的时候，另一个子数组的元素还能按照逻辑尽数被放入原数组。

快速排序:

分：随机选择枢轴元素，把枢轴与数组的第一个元素交换（固定枢轴位置，但随机枢轴大小），根据枢轴的大小，把原数组中的元素分成大于、小于、等于三个区域。

治：对大于小于枢轴的区域做递归划分

堆排序:

构建大顶堆：从最后一个叶子节点开始，对每个叶子节点都做调整大顶堆的操作

调整大顶堆：检查当前元素的左右孩子是否比自己小，若不是，则交换位置，再调整

一遍被交换子节点所在的子树。

利用大顶堆做升序排序：先构建大顶堆，再把大顶堆堆顶元素和末尾元素交换，把堆的大小减一，后调整剩余元素组成的大顶堆。

(三) 实验源代码

此部分包含主要算法部分以及相应注释（对于数据结构、数据处理与初始化、数据测试部分不再赘述）

Mergesort:

```
def merge(A,p,q,r):

    #计算左右子树的长度
    n1 = q - p + 1
    n2 = r - q
    #创建左右临时数组，并在末尾添加一个无穷大做哨兵
    L = [A[p + i] for i in range(n1)] + [float('inf')]
    R = [A[q + 1 + j] for j in range(n2)] + [float('inf')]
    # 初始化指针
    i = j = 0
    for k in range(p,r + 1):
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
def merge_sort(A,p,r):
    if p < r:
        q = (p + r)//2
        merge_sort(A,p,q)
        merge_sort(A,q + 1,r)
        merge(A, p, q, r)
```

QuickSort:

```
# Three-way Partitioning
# 将数组分为三部分：大于基准值，等于基准值，小于基准值
# 优化快排，使得算法在处理大量重复数据时的递归深度减少
```

```
# 因为普通的快排算法会将等于基准值的数排在同一侧，使得递归深度大大增加

import random
def QuickSort(A,p,r):
    if p < r:
        lt,gt = partition(A, p, r)
        QuickSort(A,p,lt - 1)
        QuickSort(A,gt + 1,r)

def partition(A,p,r):
    # 快速排序的基准固定选择最后一个元素会导致递归深度达到 O(n)
    # 触发 Python 的默认递归深度限制
    # 此处引入随机化基准选择
    pivot_index = random.randint(p,r)
    A[pivot_index],A[p] = A[p],A[pivot_index]

    pivot = A[p] #枢轴位于第一个

    lt = p #小于基准的右边界（边界不被包含）
    gt = r #大于基准的左边界（边界不被包含）
    i = p + 1 #当前元素

    while i <= gt:
        if A[i] < pivot:
            A[i],A[lt] = A[lt],A[i]
            i += 1
            lt += 1
        elif A[i] > pivot:
            A[gt],A[i] = A[i],A[gt]
            gt -= 1
            # 被交换过来的A[gt]原值还未经过比较，放置在A[i]上
            # 故i不用++
        else:#与枢轴相等的元素直接放在lt与gt之间
            i += 1
    return lt,gt
```

HeapSort:

```
#下标从0开始
# heap_size要迭代下去
#构建大顶堆
def build_max_heap(A):
    heap_size = len(A)
      #对每一个叶子都调整一遍
```

```
        for i in range(len(A)//2 -1, -1 ,-1):
            max_heapify(A, i,heap_size)
    #调整大顶堆
    def max_heapify(A,i,heap_size):
        l = 2*i + 1 # 2(i+1)-1,下标从0开始
        r = 2*i + 2 # [2(i+1) + 1] - 1

        # 注意，是 < not <=
        if l < heap_size and A[l] > A[i]:
            largest = l
        else:
            largest = i

        if r < heap_size and A[r] > A[largest]:
            largest = r

        if largest != i:
            A[largest],A[i] = A[i],A[largest]
            #递归调整被交换的子节点所在子树
            max_heapify(A,largest,heap_size)

    #利用大顶堆做数组的升序排序
    def heap_sort(A):
        build_max_heap(A)  # 第一步：构建最大堆

        heap_size = len(A)

        # 第二步：从后往前交换并调整堆
        for i in range(len(A) - 1, 0, -1):
            A[0], A[i] = A[i], A[0]  # 将最大值交换到数组末尾
            heap_size -= 1  # 缩小堆范围
            max_heapify(A, 0, heap_size)  # 调整剩余堆
```

(四)　实验数据分析对比（附数据集 URL）

#纽约出租车行程数据（NYC Taxi Trip Data）

#数据集内容：包含出租车行程的详细记录，如上车时间、车费、距离等,此测试中对 total_amount 进行排序

#URL：https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

=== 测试数据规模：5000 ===

```
数据类型          | MergeSort(秒) | QuickSort(秒) | HeapSort(秒)
-----------------------------------------------------------
随机数据          | 0.0088        | 0.0069        | 0.0086
已排序数据        | 0.0088        | 0.0061        | 0.0086
逆序数据          | 0.0087        | 0.0069        | 0.0082
重复数据          | 0.0092        | 0.0005        | 0.0060

=== 测试数据规模: 50000 ===
数据类型          | MergeSort(秒) | QuickSort(秒) | HeapSort(秒)
-----------------------------------------------------------
随机数据          | 0.1032        | 0.0888        | 0.1109
已排序数据        | 0.0953        | 0.0711        | 0.1136
逆序数据          | 0.1046        | 0.0745        | 0.1024
重复数据          | 0.0949        | 0.0048        | 0.0730

=== 测试数据规模: 500000 ===
数据类型          | MergeSort(秒) | QuickSort(秒) | HeapSort(秒)
-----------------------------------------------------------
随机数据          | 1.3148        | 0.9904        | 1.7161
已排序数据        | 1.1190        | 0.8929        | 1.4000
逆序数据          | 1.1654        | 0.9325        | 1.2917
重复数据          | 1.0819        | 0.0520        | 0.8783
```

数据分析:

快速排序在绝大多数场景下表现最优，尤其在重复数据中优势显著。归并排序表现稳定但略慢于快速排序，而 堆排序整体性能最弱。

随机数据：快速排序：通过分治策略快速分割数据，常数因子最低。归并排序：稳定但内存开销大，适合外排序或链表结构。堆排序：性能最差，仅适用于内存受限场景。

有序/逆序数据：快速排序：随机化基准避免退化，时间与随机数据接近。归并排序：性能一致，但慢于优化后的快速排序。堆排序：建堆过程对有序数据无优化，性能仍弱于前两者。

重复数据：快速排序：三向切分直接跳过重复元素，时间复杂度接近 O(n)，优势显著。归并排序：需完整遍历所有元素，无优化空间。堆排序：重复元素仍需调整堆结构，效率无明显提升。

快速排序的优势：

三向切分优化：对重复数据的处理时间复杂度接近 O(n)，显著优于归并和堆排序的 O(n log n)。

原地排序：减少内存访问开销，常数因子较低。

随机化基准选择：避免有序/逆序数据的最坏情况，时间复杂度稳定在 O(n log n)。

归并排序的稳定性：

无论数据分布如何，时间复杂度始终为 O(n log n)，但需要额外 O(n) 内存空间。

对缓存不友好，合并操作的频繁内存复制导致常数因子较高。

堆排序的劣势：

非局部内存访问：堆操作需要跳跃访问数组元素，导致缓存命中率低。

常数因子大：建堆和调整堆的过程涉及多次交换和递归，效率低于快速排序的分治策略。

二、 动态规划与贪心（综合性实验）

(一) 实验题目

1. 实验 2-S-1:0/1 背包算法大比拼。实现至少 3 种求解经典 0/1 背包问题的算法(包括贪心算法、基于递归的动态规划算法、基于循环的动态规划算法、基于回溯的算法),可选择实现附加其他约束条件的 0/1 背包问题求解算法。测试时要求能够验证不同求解算法在不同数据场景下计算结果或计算效率各有优势的特性,鼓励检索新实际工程数据集进行验证(要求标明所采用的实际数据集名称及对应 URL)。
2. 实验 2-S-2:最短路径算法大比拼。实现至少 3 种求解最短路径问题算法,可以包括同一理论算法基于不同的图数据结构(邻接矩阵或邻接链表等)的多种实现。测试时要求能够验证不同求解算法实现在不同数据场景下计算结果或计算效率各有优势

的特性,鼓励检索新实际工程数据集进行验证(要求标明所采用的实际数据集名称及对应 URL)。

(二) 实验原理

Dynamic Programming 动态规划:

“大事化小，小事化无”：本质上是对问题状态的定义和状态转移方程的定义（状态以及状态之间的递推关系）

特点:

1.把原问题分成几个相似的子问题;

2.所有的子问题都只需要解决一次;

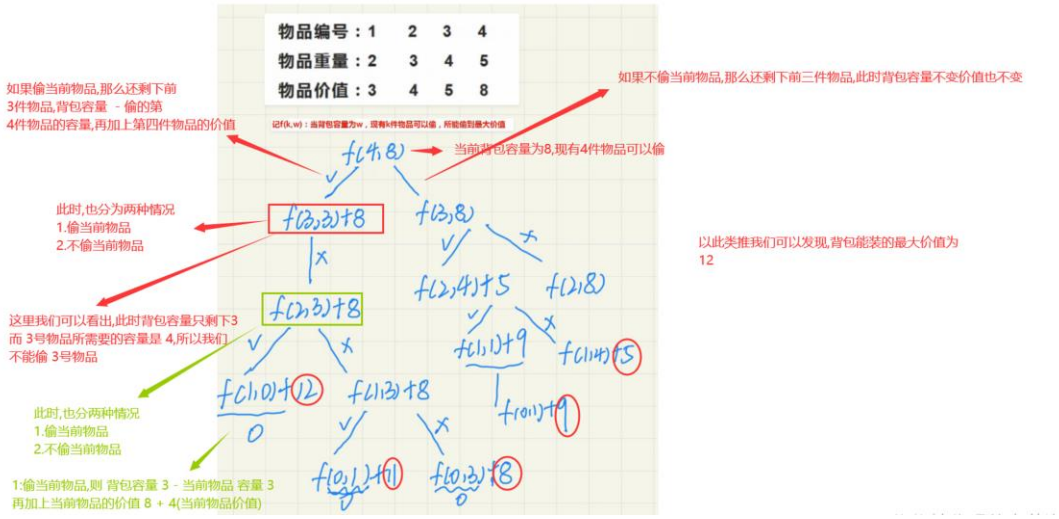
3.储存子问题的解

解题思路:

1.确定状态（状态的定义一定要形成递推关系）；2.确定状态转移方程；3.对状态做初始化；4.返回结果

0/1 背包问题:

给定 n 种物品和一个背包。物品 i 的重量是 wi,其价值为 vi，背包的容量为 C。问应如何选择物品装入背包，使得装入背包中的物品的总价值最大?在选择物品装入背包时，对每种物品 i 只有两种选择，要么装入，要么不装入，不能将物品 i 装入背包多次，也不能只装入物品 i 的一部分。因此，该问题称为 0-1 背包问题。



1.**贪心算法**——非动态规划（统揽全局），而是鼠目寸光

不适合求解 0-1 背包问题（不能得到最优解）,但是适合求解分数背包问题;

基本思想：每一步都做出**在当前看来最好的选择**——局部优化选择达到全局优化选择，但是不一定总产生优化解

贪心策略:

1.价值贪心： choose max v——背包可能载重消耗速度太快

2.重量贪心： choose min w——不能保证目标函数值的快速增加

3.**价值重量比贪心：choose max v/w (!!!!)，直到装满背包**

2.基于**递归**的**动态规划算法**

- **0-1 背包问题的 DP 策略**

$$c[i,w]=\begin{cases}0 & \text{if } i=0 \text{ or } w=0\\ c[i-1,w] & \text{if } w_i > w\\ \max(c[i-1,\ w-w_i]+v_i,\ c[i-1,\ w]) & \text{if } i>0 \text{ and } w \geq w_i\end{cases}$$

DYNAMIC PROGRAMMING FOR 0/1 KNAPSACK

```
KNAPSACK-DP(v, w, n, W)
1  for w ← 0 to W
2    do c[0,w] = 0
3  for i ← 1 to n
4    do c[i,0] ← 0
5       for w ← 1 to W
6          do if w[i] ≤ w
7                then if v[i] + c[i-1,w-w[i]] > c[i-1,w]
8                       then c[i,w] ← v[i] + c[i-1,w-w[i]]
9                       else c[i,w] ← c[i-1,w]
10               else c[i,w] ← c[i-1,w]
```

其中 C[i][j]代表，对前 i 个物品，当背包容量为 j 时，所能具有的最大价值

1. i=0/j=0，此时代表没有物品可供选择/背包没有余量可放入，则从 c[0][j]/c[i][0]值为 0

2. 否则：

if (w[i] <= j) {

//当第 i 个物品可以选择，不选择/选择-两种情况取最大值

c[i][j] = max(c[i - 1][j] , v[i - 1] + c[i - 1][j - w[i]]);}

else {

//不能选择第 i 个物品

c[i][j] = c[i - 1][j];}

3.**基于回溯的算法**

回溯法基本思想：通过**深度优先搜索**解决问题

1.开始节点：从根节点出发，这个节点是解空间的起点

2.扩展节点：在当前节点上，选择一个方向继续搜索，这个方向会形成一个新的节点

3.活节点与死节点：如果新节点有更多选择，就为活节点；如果所有选择都已经尝试，则为死节点。

4.回溯：如果当前路径不是解，回溯到上一个活节点，尝试其他选择

优化方法：

1.剪枝 1：如果当前物品的加入使得总重量超过背包重量，则停止搜索这个方向

2.剪枝 2：如果剩余物品即使全部加入，也不能超过当前已知的最优解，则停止搜索这个方向（分数背包的思想计算上限）

步骤

初始化：设置一个数组或列表来记录选择的物品。

递归函数：定义一个递归函数，接收当前物品索引、当前重量和当前价值作为参数。

边界条件：如果当前重量超过背包容量或已处理完所有物品，回溯。（剪枝 1）

选择与不选择：对于每种物品，尝试选择它或不选择它，然后递归调用函数。

更新最优解：每次找到一个解时，比较并更新已知的最优解。

递归函数：Backtrack：

1.叶子节点：i>n,新的方案，更新最优解的值

2.扩展节点：i<n,当前节点位于第 i-1 层，递归搜索子树，剪去不满足约束的节点

执行步骤：

1.计算单位价值：降序排列物品

2.从根节点出发：根节点代表当前扩展节点

3.搜索左子树（选择当前物品）：判断物品是否装入背包。

可行,更新 CP，CW，继续遍历（CP=>current price；CW=>current weight）

不可行，回溯，尝试右子树（不选当前物品）

4.计算上界：分数背包的思想

如果 bound（i）<bestp,剪枝

否则，继续搜索

最短路径问题：

采用 Dijkstra 和 BellmanFord 算法来分别实现基于邻接矩阵和邻接表的求解最短路径的算法。其中基于邻接表的 Dijkstra 算法利用优先级队列，优化了查询到已标记节点的最小距离的复杂度。而 BellmanFord 两种实现均采用剪枝策略来避免对无意义路径的探索。

Dijkstra:

贪心的思想，每次都扩展距离已标记节点距离最近的节点。不可处理带有负权边的图。

BellmanFord:可以处理带有负权边的图，也可以用于检测图中是否存在负权环。

松弛操作：对于每条边 u->v，都检查是否可以通过 u 来缩短到达路径的长度。

需要 n-1 轮松弛操作：保证对最短路径的每一条边都进行遍历——有 n 个节点的地图（无负权环），最短路径最多有 n-1 条边，故至少要松弛 n-1 次，来逐步修正最短路径。

检测负权环：在第 n 轮松弛中，若最短路径改变，则代表图中存在负权环

(三)　实验源代码

此部分包含主要算法部分以及相应注释（对于数据结构、数据处理与初始化、数据测试部分不再赘述）

Greedy：

```
vector<int> sortwv(int n, const vector<int>& w, const vector<int>& v) {
   vector<int> idx(n);//映射排序后的和排序前的物品的顺序的数组
   vector<double> u(n);//单位重量的价值（一定用 double，保证精度）
   for (int i = 0; i < n; i++) {
      if (w[i] == 0) {//物品重量为零，单位重量价值接近无穷
         u[i] = numeric_limits<double>::infinity();
      }
      else {
         u[i] = static_cast<double>(v[i]) / w[i];  // 浮点除法
      }
      idx[i] = i;//初始化映射关系
   }
//以 idx 的元素的值作为 u 数组的索引，按照 u 数组元素的值，降序排列 idx 的元素
   sort(idx.begin(), idx.end(), [&u](int a, int b) {
      return u[a] > u[b];
      });
   return idx;
}

void solve(int n, int c, const vector<int>& w, const vector<int>& v,
vector<int>& b) {
   //排序
   vector<int> idx = sortwv(n, w, v);
   //初始化
   int cur_weight = 0.0;
   fill(b.begin(), b.end(), 0);

   for (int i = 0; i < n; i++) {
      //找到单位重量价值最高的物品的原始位置索引
      int original_idx = idx[i];
      if (cur_weight + w[original_idx] <= c) {//背包能够装下当前物品
         //记录当前物品已选择
         b[original_idx] = 1;
         //更新当前总重量
         cur_weight += w[original_idx];}}}
```

DP：

```
void solve(int n, int c, const vector<int>& w, const vector<int>& v,
vector<int>& b) {
   //dp[i][j]表示在前 i 个物品中，当背包容量为 j 时的最大价值
   //i=0/j=0 时，dp 的值都为 0——没有物品可供选择/背包没有余量可放入
```

```
    vector<vector<int>> dp(n + 1, vector<int>(c + 1, 0));

    //构建DP表，直接从1开始（第一个物品 ，背包容量从1开始）
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= c; j++) {
            if (w[i - 1] <= j) {
                //当第i个物品可以选择，不选择/选择两种情况取最大值
                dp[i][j] = max(dp[i - 1][j], v[i - 1] + dp[i - 1][j - w[i -
1]]);
            }
            else {
                //不能选择第i个物品
                dp[i][j] = dp[i - 1][j];}}}

    //确定物品选择
    int remaining = c;
    for (int i = n; i >= 1; i--) {
        if (dp[i][remaining] != dp[i - 1][remaining]) {
            //代表第i个物品被选中
            b[i - 1] = 1;
            remaining -= w[i-1];}}}
```

Backtrack：

```
class Backtrack :public KnapsackSolver {
private:
    vector<int> w_sorted;  // 排序后的重量
    vector<int> v_sorted;  // 排序后的价值
    vector<int> idx_map;       // 原始索引映射
    vector<int> current_b;     // 当前选择路径
    vector<int> best_b;        // 最优选择路径
    int bestp;              // 最优价值
    int cw;                 // 当前重量
    int cp;                 // 当前价值
    int capacity;           // 背包容量

    // 排序函数同贪心算法
    vector<int> sortwv(int n, const vector<int>& w, const vector<int>& v) {
        vector<int> idx(n);
        ……
    }

    //计算上界（剩余物品的最大可能价值）,返回浮点以保留精度
```

```
    double bound(int i) {
        //左子树代表选择当前物品，计算剩余背包容量
        int left = capacity - cw;

        double bound = cp;//初始化上界为当前价值
        //在背包还有余量的时候
        while (i < w_sorted.size() && left>0) {
            if (w_sorted[i] <= left) {
                //选择当前物品，重新计算上界和剩余容量
                bound += v_sorted[i];
                left -= w_sorted[i];
            }
            else {
                //不选择当前物品，但是计算剩余物品的最大可能价值
                bound  +=  static_cast<double>(v_sorted[i])  /  w_sorted[i]  *
left;
                break;}
            i++;//看下一个物品}
        return bound;}

    //回溯核心函数
    void backtrack(int i) {
        if (i >= w_sorted.size()) {
            //递归结束，已经遍历过叶子节点
            if (cp > bestp) {
//更新最优解
                bestp = cp;
                best_b = current_b;}
            return;}

        //左子树：选择当前物品
        if (cw + w_sorted[i] <= capacity) {
            current_b[i] = 1;
            cw += w_sorted[i];
            cp += v_sorted[i];
            backtrack(i + 1);
            //回溯，看看其他的选择
            cw -= w_sorted[i];
            cp -= v_sorted[i];
            current_b[i] = 0;
        }
        //无法选择当前物品，才到这一步
        //右子树：不选择当前物品（剩余物品的浮点上界 > 整数最优值）
        if (bound(i + 1) > bestp) {
```

```
            //此时才有价值去看不选择当前物品的解（也许存在最优解）
            current_b[i] = 0;
            backtrack(i + 1);}}

    void solve(int n, int c, const vector<int>& w, const vector<int>& v,
vector<int>& b)
    {
        //按照单位价值排序
        vector<int> idx(n);
        idx = sortwv(n, w, v);
        //初始化排序后的 w/v 数组
        w_sorted.resize(n);
        v_sorted.resize(n);
        idx_map = idx;
        for (int i = 0; i < n; i++) {
            w_sorted[i] = w[idx[i]];
            v_sorted[i] = v[idx[i]];}
        //初始化状态变量
        capacity = c;
        bestp = 0;
        cw = 0; cp = 0;
        current_b.assign(n, 0);
        best_b.assign(n, 0);

        //开始回溯
        backtrack(0);

        //将排序后的结果映射回原始索引
        fill(b.begin(), b.end(), 0);
        for (int i = 0; i < n; i++) {
            if (best_b[i] == 1) {
                b[idx_map[i]] = 1;}}}
};
```

DijkstraMatrix：

```
vector<int> run(int startNode) {
    //init 数组 dist（不可达）和 visited（没到过）
    vector<int> dist(numNodes, INT_MAX);
    vector<bool> visited(numNodes, false);
    //起点到自己的距离为 0
    dist[startNode] = 0;

    for (int i = 0; i < numNodes; i++) {
```

```
        //要把所有的点都选一遍，所以要遍历 n 次
        int u = -1;
        //从未到过的点中选取最小距离的点，选择之后标记为已经过，继续循环
        for (int j = 0; j < numNodes; j++) {
            if (!visited[j] && (u == -1 || dist[j] < dist[u])) {
                //记录最小的未到达 dist 元素的索引
                u = j;}}
        if (dist[u] == INT_MAX) {
            //最短距离都是不可达，故结束
            break;}
        //否则，标记已经过
        visited[u] = true;

        for (int v = 0; v < numNodes; v++) {
            //根据邻接矩阵，在将新站点纳入后，更新 dist 数组
            if (adjMatrix[u][v] != INT_MAX && !visited[v]) {
                dist[v] = min(dist[v], dist[u] + adjMatrix[u][v]);}}}
    return dist;}
```

DijkstraList：（除了使用优先级队列管理，其余逻辑与上面的算法相似）

```
int numNodes;
//邻接表[目标节点，权重]
//邻接表储存
vector<vector<pair<int, int>>> adjList;
……
// 使用最小堆存储 {当前最短距离, 顶点}，优先队列自动维护最小值
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
pq.emplace(0, startNode);
```

BellmanFordMatrix：

```
vector<int> run(int startNode) {
    const int INF = 1e9;//防止在做加法时溢出
    vector<int> dist(numNodes, INF);
    dist[startNode] = 0;

    //保证对最短路径中的每一条边都松弛
    for (int i = 0; i < numNodes - 1; i++) {
        bool updated = false;  // 记录是否有更新
        for (int u = 0; u < numNodes; u++) {
            if (dist[u] == INF) continue; // 跳过未访问的点
            for (int v = 0; v < numNodes; v++) {
                if (adjMatrix[u][v] != INT_MAX) {
```

```
                if (dist[v] > dist[u] + adjMatrix[u][v]) {
                    dist[v] = dist[u] + adjMatrix[u][v];
                    updated = true;}}}}
        if (!updated) break; // 如果本轮无更新，提前终止
    }

    //检测负权环:如果继续松弛仍然有变化，则有负权环
    for (int u = 0; u < numNodes; u++) {
        if (dist[u] == INF) continue;
        for (int v = 0; v < numNodes; v++) {
            if (adjMatrix[u][v] != INT_MAX &&
                dist[v] > dist[u] + adjMatrix[u][v]) {

                cerr << "Graph contains negative weight cycle!" << endl;
                return vector<int>(numNodes, INT_MIN);
                // 返回 INT_MIN 表示所有节点都受负环影响;}}}
    return dist;
}
```

BellmanFordList：（除了使用邻接表储存数据，其余逻辑与上面的算法相似）

……

(四)　实验数据分析对比（附数据集 URL）

背包问题测试数据分析：

```
=== Test Case 1 (n=5, c=100) ===
[Greedy Algorithm]

Algorithm: Greedy Algorithm
Total Value: 109 (Expected: 133)
Total Weight: 51/100
Status: FAIL
Time: 8.5e-06s
[Dynamic Programming]

Algorithm: DPKnapsack
Total Value: 133 (Expected: 133)
Total Weight: 79/100
Status: PASS
Time: 5.83e-05s
[Backtrack]

Algorithm: Backtrack
Total Value: 133 (Expected: 133)
```

```
Total Weight: 79/100
Status: PASS
Time: 8.9e-06s
=== Test Case 9 (n=100, c=1000) ===
[Greedy Algorithm]

Algorithm: Greedy Algorithm
Total Value: 2844 (Expected: 2852)
Total Weight: 993/1000
Status: FAIL
Time: 4.89e-05s
[Dynamic Programming]

Algorithm: DPKnapsack
Total Value: 2852 (Expected: 2852)
Total Weight: 999/1000
Status: PASS
Time: 0.0042185s
[Backtrack]

Algorithm: Backtrack
Total Value: 2852 (Expected: 2852)
Total Weight: 999/1000
Status: PASS
Time: 0.0001201s
```

贪心算法:

优点: 运行时间极短（微秒级）。

缺点: 在大多数测试用例中无法达到最优解（如 Test Case 1、3、4-9），验证了其在 0-1 背包问题中的局限性（局部最优解）。

典型问题: 无法处理需要精确组合物品的场景（如 Test Case 1 的预期值 133 与贪心结果 109 的差距）。

动态规划（DP）:

优点: 在所有测试用例中均正确，验证了其可靠性和准确性。

性能: 运行时间随规模增长而增加（如 Test Case 4-9 中约 0.005 秒），但在合理范围内。时间复杂度: O(nW)。

适用性：适合中小规模数据（n ≤ 1000，容量 ≤ 1e4）。

回溯法：

优点：在所有测试用例中均正确，且运行时间接近动态规划（微秒级）。

异常表现：理论上时间复杂度为 O(2n)O(2n)，但在实际测试中表现出高效性（如 n=100 时仅需 0.0001 秒）。

关键原因：通过单位价值排序和上界剪枝，大幅减少搜索路径。回溯法运行时间接近动态规划，表明剪枝策略成功淘汰了绝大多数无效路径。

最短路径测试数据分析：

数据集设计与生成方法

1. 超稀疏图（邻接表优势场景）

- 特征：10,000 节点，20,000 条边（平均每个节点仅 2 条边）
    - 生成脚本：

```
import numpy as np

def generate_sparse_large(filename):
    nodes = 10000
    edges = 20000
    with open(filename, 'w') as f:
        f.write(f"{nodes} {edges}\n")
        for _ in range(edges):
            src = np.random.randint(0, nodes)
            dest = np.random.randint(0, nodes)
            weight = np.random.randint(1, 10)
            f.write(f"{src} {dest} {weight}\n")

generate_sparse_large("data/sparse_large.txt")
```

2. 超稠密图（邻接矩阵优势场景）

- 特征：500 节点，249,500 条边（完全图，每个节点连接其他 499 节点）

- 生成脚本:

```
def generate_dense_large(filename):
    nodes = 500
    edges = nodes * (nodes - 1)  # 500*499 = 249,500
    with open(filename, 'w') as f:
        f.write(f"{nodes} {edges}\n")
        for src in range(nodes):
            for dest in range(nodes):
                if src != dest:
                    weight = np.random.randint(1, 10)
                    f.write(f"{src} {dest} {weight}\n")

generate_dense_large("data/dense_large.txt")
```

测试结果：

```
=== Testing Scenario: Sparse Graph (10k nodes, 20k edges) ===
Algorithm: Dijkstra (Matrix)
Average Time: 0.0109262s
Distance to Node 100: 2147483647
------------------
Algorithm: Dijkstra (List)
Average Time: 0.00013292s
Distance to Node 100: 2147483647
------------------
Algorithm: Bellman-Ford (Matrix)
Average Time: 0.00356713s
Distance to Node 100: 1000000000
------------------
Algorithm: Bellman-Ford (List)
Average Time: 0.0002s
Distance to Node 100: 1000000000
------------------

=== Testing Scenario: Dense Graph (500 nodes, 249k edges) ===
Algorithm: Dijkstra (Matrix)
Average Time: 0.116494s
Distance to Node 100: 2
------------------
Algorithm: Dijkstra (List)
Average Time: 0.00314816s
Distance to Node 100: 2
------------------
Algorithm: Bellman-Ford (Matrix)
```

```
Average Time: 0.021695s
Distance to Node 100: 2
------------------
Algorithm: Bellman-Ford (List)
Average Time: 0.00889717s
Distance to Node 100: 2
------------------
```

1. **稀疏图场景（10k 节点，20k 边）**

   Dijkstra 算法：

   邻接矩阵：平均时间 0.0109 秒。

   邻接表：平均时间 0.00013 秒，性能显著优于邻接矩阵（约快 84 倍），因邻接表仅遍历实际存在的边，而邻接矩阵需遍历所有可能的边组合（10k×10k）。

   关键优化：邻接表版本可能实现了优先队列优化（时间复杂度从 O(V2)O(V2) 优至 O(E+VlogV)O(E+VlogV)）。

   Bellman-Ford 算法：

   邻接矩阵：平均时间 0.0035 秒。

   邻接表：平均时间 0.0002 秒，性能优于邻接矩阵（快 17 倍），因邻接表跳过了无效边。

2. **稠密图场景（500 节点，249k 边）**

   Dijkstra 算法：

   邻接矩阵：平均时间 0.116 秒，节点 100 的距离为 2。

   邻接表：平均时间 0.0031 秒（快 37 倍），显著优于邻接矩阵，可能因优先队列优化大幅减少无效操作。

   Bellman-Ford 算法：

   邻接矩阵：平均时间 0.021 秒，节点 100 的距离为 2。

   邻接表：平均时间 0.0089 秒（快 2.4 倍），因邻接表跳过了无效边（如 dist[u] == INF 的节点）。

   结果正确性：所有算法输出一致（距离为 2），验证稠密图中最短路径计算的正确性。

三、 常规搜索（综合性实验）

(一) 实验题目

实验 3-S-1:**皇后问题**算法大比拼。针对皇后问题求所有可行解与只求 1 个可行解两种场景,分别各实现至少 2 种求解算法,可选择使用不同遍历方式实现,如 DFS 或 BestFS 等,也可以对限界函数或 BestFS 中不同的 Best 标准进行不同方式的实现或改进。测试时要求能够验证同一场景下不同求解算法在计算效率上的差异。

(二) 实验原理

皇后问题：

在 N*N 的棋盘上放置 N 个皇后，使得它们互相不攻击——不在同一行、列或者主副对角线上。

基于不同的遍历方式：DFS 与 BestFS；分别实现基于递归的深度优先 DFS 的回溯算法和基于优先级队列的启发式搜索 BestFS （用剪枝策略代替显示回溯）。

DFS 中：从棋盘的第一行开始，逐行尝试在每一列放置皇后。每次放置后，立即递归进入下一行继续尝试，**深度优先**探索路径。回溯：在 DFS 中，若在某一行的每一列都无法合法放置皇后，则回溯到上一行，重新调整此行上皇后的位置。

BestFS 中：根据评估函数和已经放置皇后的情况，对未放置皇后的行，评估在每一列上放置皇后的分数。将分数通过优先级队列管理，每次扩展评分高的节点。评估函数中的 Best 标准，即启发式规则分别实现两种：最少冲突数优先（**选择下一步中导致冲突最少得列**）和剩余可行位置最多（**优先扩展剩余行中可行列较多的分支**）。过程中还实现了直接剪除非法分支的冲突检查剪枝和避免重复扩展的状态去重剪枝，从而代替显示的回溯。

(三)　实验源代码

此部分包含主要算法部分以及相应注释（对于数据结构、数据处理与初始化、数据测试部分不再赘述）

SingelDFS:

```
//检查当前行能否在第 col 列放置 queen
bool isValid(int row, int col) {
    for (int i = 0; i < row; i++) {
        //不可在同一行、列和主副对角线上放置皇后
        if (col == queens[i] || abs(row - i) == abs(col - queens[i])) {
            //不可放置
            return false;
        }
    }
    //可以放置
```

```
    return true;
}
//深度优先遍历
bool dfs(int row, int n) {
    if (row == n) {
        //已经把 n 行的皇后全部放置成功
        found = true;
        return true;//已经找到解
    }
    for (int col = 0; col < n; col++) {
        if (isValid(row, col)) {
            queens[row] = col;
            if (dfs(row + 1, n))//递归到下一行
                return true;//此行放置成功
            queens[row] = -1;//否则回溯
        }
    }
    return false;//没有放置成功}
```

AllDFS:

```
void dfs(int row, int n, vector<int>& queens)
{
    if (row == n) {
        solutions.push_back(queens);//把当前的解加入解集
        return;
    }

    for (int col = 0; col < n; col++) {
        if (isValid(row, col, queens)) {
            queens[row] = col;
            dfs(row + 1, n, queens);
            queens[row] = -1;//回溯，因为要找到所有的解
        }
    }
}
```

SingleDFS:

```
    //评估函数：计算 score（剩余冲突数，所以分数越高反而优先级越低）
    int heuristic(const vector<int>& state, int n) {
        int conflicts = 0;
        //当前行（从 0 开始）
        int row = state.size();
```

```
            //遍历剩余的所有行，计算冲突数
            for (int r = row; r < n; r++) {
                for (int c = 0; c < n; c++) {
                    bool valid = true;

                    //检查在（r，c）处放置是否与当前状态冲突
                    for (int i = 0; i < state.size(); i++) {
                        if (state[i] == c || abs(r - i) == abs(c - state[i])) {
                            valid = false;
                            break;}}
                    if (!valid) conflicts++;}}
            return conflicts;
        }
        vector<int> solveSingleBestFS(int n) {
            //优先级队列管理被评估过的位置
            priority_queue<Node> pq;
            //初始化优先级队列
            pq.push(Node({}, heuristic({}, n)));

            unordered_set<string> visited;//用于去重，避免重复状态扩展
            visited.insert("");//初始状态已访问
            while (!pq.empty()) {
                //利用优先级队列选择优先级高的位置来扩展
                Node node = pq.top();
                pq.pop();
                int row = node.state.size();//计算当前行

                //找到解
                if (row == n) return node.state;
                //遍历当前行的每一列
                for (int col = 0; col < n; col++) {
                    bool valid = true;//预设可行性
                    //对已经存在过的状态，判断当前位置是否可行
                    for (int i = 0; i < row; i++) {
                        if (node.state[i] == col || abs(row - i) == abs(col -
node.state[i])) {        //产生冲突
                            valid = false;
                            break;//直接跳出
                        }
                    }

                    if (valid) {
                        //仅扩展合法分支
                        vector<int> new_state = node.state;
```

```
                    new_state.push_back(col);
                    int score = heuristic(new_state, n);

                    //将新状态加入队列，并进行去重
                    string state_str = "";

                    //把 new_state 中表示已经放置好的位置转化成字符串
                    for (int val : new_state) state_str += to_string(val) +
","
                    //如果 new_state 与 visited 中的无重复
                    if (visited.find(state_str) == visited.end()) {
                        pq.push(Node(new_state, score));
                        visited.insert(state_str);}}}}
            return {};//无解
        }
```

AllBestFs:

```
        //计算 score（剩余可行列的数量，所以评分越高优先级越高）
        int heuristic(const vector<int>& state, int n) {
            int row = state.size();
            int available = 0;
            //score 只计算当前行的可行的列
            for (int col = 0; col < n; col++) {
                int valid = true;
                for (int i = 0; i < row; i++) {
                    if (state[i] == col || abs(row - i) == abs(col - state[i]))
{
                        valid = false;
                        break;}}
                if (valid) available++;}
            return available;}

        vector<vector<int>> solveAllBestFS(int n)
        {
            priority_queue<Node> pq;
            pq.push(Node({}, heuristic({}, n)));

            vector<vector<int>> solutions;
            unordered_set<string> visited;

            while (!pq.empty()) {
                Node node = pq.top();
                pq.pop();
```

```
            int row = node.state.size();
            //找到了一个解
            if (row == n) {
                solutions.push_back(node.state);//把当前解放入解集
                continue;//继续搜索其他解
            }

            for (int col = 0; col < n; col++) {
                int valid = true;

                for (int i = 0; i < row; i++) {
                    if (node.state[i] == col || abs(row - i) == abs(col -
node.state[i])) {
                        valid = false;
                        break;}}

                if (valid) {
                    vector<int> new_state = node.state;
                    new_state.push_back(col);
                    int score = heuristic(new_state, n);

                    //进行去重，避免重复搜索相同状态
                    string state_str = "";
                    for (int val : new_state) state_str += to_string(val) +
",";

                    if (visited.find(state_str) == visited.end()) {
                        pq.push(Node(new_state, score));
                        visited.insert(state_str);}}}}
        return solutions;}
```

(四)　实验数据分析对比

```
Testing N=8 for different algorithms:

Single
SingleDFS (Single Solution) for n=8 took 0ms.
SingleBestFS (Single Solution) for n=8 took 1ms.

All
AllDFS (All Solutions) for n=8 took 0ms.
AllBestFS (All Solutions) for n=8 took 25ms.

Testing N=12 for a larger problem:
```

```
Single
SingleDFS (Single Solution) for n=12 took 0ms.
SingleBestFS (Single Solution) for n=12 took 54ms.

All
AllDFS (All Solutions) for n=12 took 558ms.
AllBestFS (All Solutions) for n=12 took 16544ms.
```

单解场景:

N=8：DFS (0ms)：DFS 逐行放置皇后，通过回溯快速找到第一个解。由于棋盘较小，搜索路径短，几乎无冗余计算。BestFS (1ms)：启发式函数需计算剩余冲突数，优先队列操作引入额外开销。尽管减少搜索范围，但小规模问题中开销抵消了优势。

N=12：DFS (0ms)：DFS 在单解场景下仍能快速找到解，路径深度仅 12 步，冲突检查高效。BestFS (54ms)：启发式函数需遍历剩余所有行（12 行）的冲突，计算复杂度为 $O(N^2)$，导致评分耗时增加。优先队列节点数增多，插入/弹出操作变慢。

所以，单解场景下，DFS 更优，回溯剪枝直接，无额外计算，适合快速找到首个解。

BestFS 的启发式函数和队列管理开销显著，尤其在大 N 时。

全解场景:

N=8：DFS (0ms)：全解数为 92，回溯剪枝高效，仅需遍历合法路径。BestFS (25ms)：需扩展所有合法节点，频繁计算启发式评分和队列操作，效率低于 DFS。

N=12：DFS (558ms)：需遍历所有 12 种可能路径，但回溯剪枝大幅减少实际搜索量。

BestFS (16544ms): 启发式无法有效剪枝全解搜索, 维护优先队列和去重导致开销剧增。

所以，全解场景下，DFS 更优，回溯天然适合穷举所有解，无冗余数据结构。

BestFS 在启发式上有误导，评分函数为单解设计，全解时无法有效引导。节点数指数增长，优先队列操作耗时。去重开销中哈希表操作和字符串转换拖慢速度。