算法合集

目录

一、动态规划

1. 定义与示例

动态规划算法（Dynamic Programming，DP）是一种解决多阶段决策问题的数学思想，通常用于优化问题的求解。它的核心思想是将原问题分解成子问题，通过记录每个子问题的最优解来求得原问题的最优解。该算法具有高效、优化、可行性等特点，广泛应用于计算机科学、运筹学、经济学、生物学等领域。

时间复杂度与空间复杂度：

动态规划算法的时间复杂度和空间复杂度通常是根据具体问题和算法实现而定的。一般来说，动态规划算法的时间复杂度和空间复杂度都与问题的规模和状态数有关。对于一个具有 n 个状态和 m 个决策的动态规划问题，它的时间复杂度通常为 $O(nm)$ 或 $O(nm\log m)$，其中 $m\log m$ 是对状态进行排序的复杂度。

对于空间复杂度，通常情况下需要使用一个二维数组来存储每个状态的最优解，因此空间复杂度为 $O(nm)$。

需要注意的是，在某些情况下，动态规划算法可能会使用滚动数组等优化技术来减少空间复杂度，从而将空间复杂度降至 $O(m)$ 或 $O(n)$ 等更低的级别。

以**背包问题**为例，假设有一个容量为 W 的背包，以及 n 个物品，第 i 个物品重量为 wi，价值为 vi。现在需要选择一些物品放入背包中，使得背包中物品的总重量不超过 W，同时总价值最大。该问题可以用动态规划算法求解。

首先定义一个二维数组 dp，其中 dp[i][j]表示在前 i 个物品中选择若干个物品放入容量为 j 的背包中，能够得到的最大总价值。由于每个物品只能选择一次，因此 dp[i][j]的值只能从以下两种情况中取最大值：

不选第 i 个物品，则 dp[i][j] = dp[i-1][j]

选第 i 个物品，则 dp[i][j] = dp[i-1][j-wi] + vi

因此, 对于每个 i 和 j, 可以通过比较以上两个值来更新 dp[i][j]。最终, dp[n][W] 即为问题的最优解。

动态规划算法通过将原问题分解成多个子问题来求解，避免了重复计算，提高了效率，因此被广泛应用于各种优化问题的求解。

2. 具体运行过程

动态规划算法的具体运行过程如下：

1. 定义状态：将原问题拆解成若干个子问题，并定义状态表示子问题的解，一般用一个二维数组 dp 来表示, 其中 dp[i][j]表示在子问题 i 中达到状态 j 的最优解。
2. 初始化状态：对于每个子问题 i 的初始状态，需要对 dp[i][j]进行初始化，即使得 dp[i][j]的值满足问题的边界条件。例如, 在背包问题中，当容量为 0 时，背包能够装入的物品价值为 0，因此 dp[i][0]=0。
3. 状态转移方程：通过子问题之间的递推关系，计算出每个子问题的最优解，即状态转移方程。根据问题的不同，状态转移方程也有所不同。例如，在背包问题中，状态转移方程为 dp[i][j]=max(dp[i-1][j], dp[i-1][j-wi]+vi)，表示在前 i 个物品中选择若干个物品放入容量为 j 的背包中，能够得到的最大总价值。
4. 求解目标：根据最终状态，得到原问题的最优解。例如，在背包问题中，背包能够装下的最大价值即为 dp[n][W]，其中 n 为物品的数量，W 为背包的容量。

二、贪心算法

1. 定义与示例

贪心算法（Greedy algorithm）是一种解决最优化问题的算法，它通过在每一步选择中都采取当前状态下最优的选择，最终得到全局最优解。贪心算法通常比动态规划算法更加简单和高效，但是它并不能保证一定能够得到最优解。

贪心算法的基本思路是**从问题的某一个初始解出发，通过局部最优的选择来构造问题的最优解**。在每一步中，贪心算法**选择当前状态下最优的选择，而不考虑未来的后果**。由于贪心算法只考虑当前状态下的最优解，因此它可以在很短的时间内得到解决方案。

举个例子，假设你有一袋糖果，需要将糖果分配给两个小孩。第一个小孩喜欢巧克力，第二个小孩喜欢糖果。现在，你需要决定如何分配糖果才能让两个小孩都满意，并且让分配出的糖果总数最大。

一种贪心算法的思路是，每次将一颗巧克力和一颗糖果一起分配给两个小孩，直到没有更多的糖果可以分配。这个算法的思路是每次都选择当前状态下的最优解，即选择一颗巧克力和一颗糖果，以满足两个小孩的需求，并且让分配出的糖果总数最大。

代码：

```
package main

import "fmt"

func candyDistribution(candies []int) int {
    n := len(candies)
    if n < 2 {
        return n
    }
    // 将糖果按照大小排序
    sort.Ints(candies)
    // 初始化小孩分配的糖果数
    count := 1
    pre := 0 // 记录上一个分配的糖果
    // 遍历糖果，依次分配给小孩
    for i := 1; i < n; i++ {
        // 如果这颗糖果可以分配给当前小孩，并且它的大小不等于上一颗分配
```

```
的糖果的大小
            if candies[i] > candies[pre] && candies[i] != candies[pre] {
                // 分配给当前小孩
                count++
                // 更新上一个分配的糖果
                pre = i
            }
        }
        // 返回分配的糖果数
        return count
}

func main() {
        candies := []int{1, 2, 2, 3, 4, 4, 5}
        fmt.Println(candyDistribution(candies)) // 输出：5
}
```

三、二分查找算法

1. 定义与示例

二分查找算法，也叫折半查找算法，是一种在有序数组中查找特定元素的高效算法。

二分查找的基本思想是，对于有序数组，通过比较中间位置元素与目标元素的大小关系，将查找范围缩小一半，逐步缩小查找范围，直到找到目标元素或者确定目标元素不存在为止。

二分查找算法的具体实现步骤如下：

首先确定数组的左边界 $left$ 和右边界 $right$，初始时 $left=0$，$right=n-1$，其中 n 为数组长度。

取中间位置 $mid=\lfloor (left+right)/2\rfloor$，其中 $\lfloor \cdot \rfloor$ 表示向下取整。

比较中间位置元素 $nums[mid]$ 与目标元素 $target$ 的大小关系，如果 $nums[mid]>target$，则在左半部分继续查找，更新右边界 $right=mid-1$；如果 $nums[mid]<target$，则在右半部分继续查找，更新左边界 $left=mid+1$；如果 $nums[mid]=target$，则直接返回 mid，找到目标元素。

重复步骤 2 和步骤 3，直到找到目标元素或者确定目标元素不存在为止。

如果经过以上步骤查找完整个数组，仍未找到目标元素，则表示目标元素不存在于数组中。

二分查找算法的时间复杂度为 $O(\log n)$，其中 n 表示数组长度。由于每次查找都将查找范围缩小一半，因此最多需要进行 $\log n$ 次比较即可找到目标元素，具有较高的查找效率。

2. 代码

```
func binarySearch(nums []int, target int) int {
    left, right := 0, len(nums)-1
    for left <= right {
        mid := (left + right) / 2
        if nums[mid] == target {
            return mid
        } else if nums[mid] > target {
            right = mid - 1
        } else {
            left = mid + 1
        }
    }
    return -1
}
```

四、哈希查找算法

1. 定义与示例

哈希查找（Hash Search）是一种利用哈希表（Hash Table）实现的查找算法，它可以快速地在一组数据中查找指定元素。哈希表是一种基于散列函数（Hash Function）实现的数据结构，可以将不同的元素映射到不同的位置，因此可以实现高效的元素查找。

哈希查找的基本思路是，首先将待查找的元素通过散列函数映射到哈希表中的某个位置，然后在该位置上查找是否存在该元素。如果该位置上没有该元素，则表示该元素不存在于哈希表中；否则，就找到了该元素。

哈希查找的优点是查找速度快，时间复杂度通常为 O(1)（最坏情况下也只有 O(n)）。但是它也存在一些缺点，比如需要解决哈希冲突（Hash Collision）问题，即多个元素可能映射到同一个位置的情况，解决哈希冲突的方法有很多种，如链地址法、开放地址法等。

2. 代码

下面是一个简单的哈希查找的实现示例，假设有一组数据 nums 和待查找的元素 target：

```
func hashSearch(nums []int, target int) bool {
    hashTable := make(map[int]bool)
    for _, num := range nums {
        hashTable[num] = true
    }
    return hashTable[target]
}
```

该函数首先定义一个哈希表 hashTable，使用 Go 语言的 map 类型实现，然后遍历 nums 数组中的所有元素，将它们映射到哈希表中。最后，检查 target 是否在哈希表中即可，返回 true 或 false 表示目标元素是否存在于 nums 数组中。由于使用了哈希表，查找的时间复杂度为 O(1)。

五、冒泡排序算法

1. 定义与示例

冒泡排序（Bubble Sort）是一种简单的排序算法，它的基本思想是依次比较相邻的元素，如果它们的顺序不对就交换它们的位置，直到没有任何一对元素需要交换为止。冒泡排序的名称由于排序过程中较小的元素会逐渐“冒泡”到序列的前面而得名。

具体实现时，冒泡排序的算法流程如下：

比较相邻的两个元素。如果第一个比第二个大，就交换它们的位置；

对每一对相邻的元素进行同样的比较操作，从第一对到最后一对，这样一轮下来，最后一个元素就会是序列中的最大值；每一轮下来就会选出一个最大的元素，排在队列的最末尾。

重复执行上述操作，直到整个序列按照从小到大的顺序排列完成。

冒泡排序算法的时间复杂度为 O(n^2)，空间复杂度为 O(1)。

2. 代码

```
func bubbleSort(nums []int) []int {
    n := len(nums)
    for i := 0; i < n; i++ {
        for j := 0; j < n-i-1; j++ {
            if nums[j] > nums[j+1] {
                nums[j], nums[j+1] = nums[j+1], nums[j]
            }
        }
    }
    return nums
}
```

六、快速排序算法

1. 定义与示例

快速排序（Quick Sort）是一种常用的排序算法，它的基本思想是通过分治的思想将待排序的序列分成两个子序列，其中一个子序列的所有元素都比另一个子序列的所有元素小，然后再递归地对子序列进行排序，直到整个序列有序。

具体实现时，快速排序的算法流程如下：

1. 选取一个基准元素（pivot）；
2. 将序列中所有比基准元素小的元素放到基准元素的左边，所有比基准元素大的元素放到基准元素的右边；
3. 递归地对基准元素左右两个子序列进行排序，直到整个序列有序。

快速排序算法的时间复杂度通常为 O(nlogn)，但最坏情况下的时间复杂度为 O(n^2)，空间复杂度为 O(logn)。

2. 代码

```
func quickSort(arr []int) []int {
    if len(arr) <= 1 {
        return arr
    }
    pivot := arr[0]
    var left, right []int
    for i := 1; i < len(arr); i++ {
        if arr[i] < pivot {
            left = append(left, arr[i])
        } else {
            right = append(right, arr[i])
        }
    }
    left = quickSort(left)
    right = quickSort(right)
```

```
    return append(append(left, pivot), right...)
}
```

1. 如果传入的切片长度小于等于 1，则直接返回该切片，因为只有一个元素或者没有元素时，不需要排序。
2. 将第一个元素设为基准值 pivot，用 left 和 right 两个空切片来存放比 pivot 小和大的元素。
3. 遍历除了第一个元素以外的所有元素，将比 pivot 小的元素放入 left 切片，将比 pivot 大的元素放入 right 切片。
4. 对 left 和 right 两个切片进行递归排序, 直到 left 和 right 的长度都小于等于 1，返回 left 和 right 切片。
5. 将 left 切片、pivot 和 right 切片按顺序合并起来，返回合并后的切片，这个切片就是已经排好序的切片。

需要注意的是，这个实现方式不太稳定，如果遇到特定的数据集合，比如大量重复的数据，就容易出现栈溢出等问题。正常情况下，建议使用优化过的快速排序算法实现。

七、二叉树的深度遍历算法

1. 前序遍历定义与示例

二叉树的深度遍历算法，主要包括三种遍历方式，分别是前序遍历、中序遍历和后序遍历。下面我分别介绍一下这三种遍历方式。

前序遍历

前序遍历的遍历顺序是先遍历当前节点, 然后遍历左子树, 最后遍历右子树。其遍历方式可以用以下步骤表示：

访问当前节点。

如果当前节点有左子树，则递归遍历左子树。

如果当前节点有右子树，则递归遍历右子树。

2. 前序遍历代码

```
type TreeNode struct {
    Val int
    Left *TreeNode
    Right *TreeNode
}

func preorderTraversal(root *TreeNode) []int {
    if root == nil {
        return nil
    }
    res := []int{root.Val}
    left := preorderTraversal(root.Left)
    right := preorderTraversal(root.Right)
    res = append(res, left...)
    res = append(res, right...)
    return res
}
```

3. 中序遍历定义与示例

中序遍历的遍历顺序是先遍历左子树, 然后遍历当前节点, 最后遍历右子树。其遍历方式可以用以下步骤表示：

如果当前节点有左子树，则递归遍历左子树。

访问当前节点。

如果当前节点有右子树，则递归遍历右子树。

```
func midTraversal(root *Node) []string    {
    if root == nil{
        return nil
    }
    res := midTraversal(root.left)
```

```
    res = append(res, root.data)
    right := midTraversal(root.right)
    res = append(res, right...)
    return res
}
```

4. 后序遍历定义与示例

后序遍历的遍历顺序是先遍历左子树和右子树，最后遍历当前节点。其遍历方式可以用以下步骤表示：

如果当前节点有左子树，则递归遍历左子树。

如果当前节点有右子树，则递归遍历右子树。

访问当前节点。

```
func afterTraversal(root *Node) []string    {
    if root == nil{
        return nil
    }
    res := afterTraversal(root.left)
    right := afterTraversal(root.right)
    res = append(res,right...)
    res = append(res, root.data)
    return res
}
```

八、二叉树的广度遍历算法

1. 定义与示例

二叉树的广度遍历（BFS）算法是一种按层级顺序遍历二叉树的算法。其基

本思想是从根节点开始，依次按照从左到右的顺序遍历每一层节点，直到遍历到最后一层。

广度遍历算法通常使用队列来实现，具体过程如下：

将根节点入队；

取出队首节点，将其值存储到结果集中；

将队首节点的左右子节点（如果存在）依次入队；

重复步骤 2 和步骤 3，直到队列为空。

2. 代码

```
type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}
func bfs(root *TreeNode) []int {
    if root == nil {
        return []int{}
    }
    var res []int
    queue := []*TreeNode{root}

    for len(queue) > 0 {
        node := queue[0]
        queue = queue[1:]
        res = append(res, node.Val)
        if node.Left != nil {
            queue = append(queue, node.Left)
        }
        if node.Right != nil {
            queue = append(queue, node.Right)
```

```
            }
        }
        return res
    }
```

使用了一个队列来存储每一层的节点，并且使用了一个切片来存储遍历的结果。在遍历过程中，我们不断从队列中取出节点，将其值存储到结果集中，并将其左右子节点依次入队。最终返回结果集即可。