Main Page - Algorithms for Competitive Programming --- 主页 - 竞赛编程算法 (cp-algorithms.com)

2022年11月26日 12:26

动态规划——分而治之

2022年10月9日 9:17

θ= (2x + 10)/(x+10)

y = 1/2θ(10² - x²) = 1/2(2x+10)/(x+10)(10² - x²)
= (x+5)/(x+10)(10²-x²)
= (x+5)/(x+10)(10-x)(10+x)
= (x+5)(10-x)
= 10x+x²+50-5x
= -x²+5x+50
= -(x-5/2)²+225/4

Divide and Conquer is a dynamic programming optimization.

分而治之是一种动态规划优化。

Preconditions 前提条件

Some dynamic programming problems have a recurrence of this form:

一些动态规划问题具有这种形式的重现：

$$dp(i,j) = \min_{0 \le k \le j} dp(i-1,k-1) + C(k,j)$$

where $C(k,j)$ is a cost function and $dp(i,j) = 0$ when $j < 0$.

其中 $C(k,j)$ 是成本函数， $dp(i,j) = 0$ 是 $j < 0$ 。

注	在动态规划中，**成本函数**通常称为「状态转移方程」或「递推关系」。它描述了问题的最优子结构，即如何通过已解决的子问题来求解更大规模的原问题。动态规划的核心思想是将原问题划分成若干个重叠子问题，并且通过存储子问题的解来避免重复计算，从而提高算法的效率。 成本函数通常以递归的形式定义，并且用于表示问题的最优解与其子问题最优解之间的关系。在动态规划中，我们使用一个「状态」来表示问题的规模或特征，并将问题的解定义为这个「状态」的函数。成本函数描述了如何从一个状态转移到另一个状态，通常使用数学符号或代码来表示。 动态规划的一般步骤如下： 1. 定义状态：确定问题的状态，即表示问题规模或特征的变量。状态的选择通常是为了保证子问题的重叠性。

2. 确定初始状态：找出问题规模最小的状态，即基本问题的解，作为动态规划的初始状态。

3. 定义成本函数（状态转移方程）：描述如何从一个状态转移到另一个状态，通常通过递归或迭代的方式表示。

4. 确定计算顺序：确定状态之间的计算顺序，通常通过自底向上的方式计算，先计算小规模的状态，然后逐步计算更大规模的状态，直至求解原问题。

5. 计算最优解：根据成本函数计算得出最优解，通常是从初始状态逐步迭代到目标状态。

成本函数的设计在动态规划中非常关键，不同的问题可能会有不同的成本函数。**在一些问题中，成本函数可能只是一个简单的递推式；而在其他问题中，成本函数可能会涉及到更复杂的状态转移过程和条件。**因此，对于不同的动态规划问题，需要仔细分析问题的性质，选择合适的状态和成本函数，以获得高效的动态规划算法。

当谈到动态规划时，一个经典的例子就是「斐波那契数列」。斐波那契数列是一个数列，其中每个数都是前两个数之和，起始数字通常为0和1。

数列的前几个数字是：0, 1, 1, 2, 3, 5, 8, 13, 21, ...

我们来看看如何使用动态规划来解决计算斐波那契数列的问题。

1. 定义状态：在这个例子中，我们可以用一个整数 n 表示问题的状态，表示我们要计算第 n 个斐波那契数。

2. 确定初始状态：由于斐波那契数列的定义，初始状态是已知的：fib(0) = 0 和 fib(1) = 1。

3. 定义成本函数（状态转移方程）：我们可以使用递归的方式来定义成本函数，即用已经计算出的前两个斐波那契数求解当前数：

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

这是一个简单的递归实现，但它存在重复计算的问题，因为在计算 `fibonacci(n)` 时会重复计算 `fibonacci(n-1)` 和 `fibonacci(n-2)`。

4. 确定计算顺序：为了避免重复计算，我们可以使用动态规划的自底向上方法，从最小规模的问题开始，逐步计算更大规模的问题。

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1

    dp = [0] * (n + 1)
```

```
    dp[0] = 0
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

在这个实现中，我们使用一个列表 `dp` 来存储已经计算过的斐波那契数，从而避免重复计算，而是直接使用已有的结果来计算新的斐波那契数。

现在，我们可以调用 `fibonacci(n)` 函数来计算斐波那契数列中第 n 个数了。例如，`fibonacci(5)` 将返回 `5`，`fibonacci(10)` 将返回 `55`。这个方法在计算大规模斐波那契数时，效率明显优于简单的递归实现。

字符串哈希

2023年7月24日 9:15

Hashing algorithms are helpful in solving a lot of problems.

哈希算法有助于解决很多问题。

We want to solve the problem of comparing strings efficiently. The brute force way of doing so is just to compare the letters of both strings, which has a time complexity of $O(\min(n_1, n_2))$ if n_1 and n_2 are the sizes of the two strings. We want to do better. The idea behind the string hashing is the following: we map each string into an integer and compare those instead of the strings. Doing this allows us to reduce the execution time of the string comparison to $O(1)$.

我们想要解决高效比较字符串的问题。这样做的强力方法只是比较两个字符串的字母，如果 n_1 和 n_2 是两个字符串的大小，则时间复杂度为 $O(\min(n_1, n_2))$ 。我们想要做得更好。字符串散列背后的想法如下：我们将每个字符串映射为一个整数，然后比较它们而不是字符串。这样做可以减少与 $O(1)$ 进行字符串比较的执行时间。

For the conversion, we need a so-called **hash function**. The goal of it is to convert a string into an integer, the so-called **hash** of the string. The following condition has to hold: if two strings s and t are equal ($s = t$), then also their hashes have to be equal ($\text{hash}(s) = \text{hash}(t)$). Otherwise, we will not be able to compare strings.

为了进行转换，我们需要一个所谓的哈希函数。它的目标是将字符串转换为整数，即所谓的字符串的哈希。必须满足以下条件：如果两个字符串 s 和 t 相等 ($s = t$)，那么它们的哈希值也必须相等 ($\text{hash}(s) = \text{hash}(t)$)。否则，我们将无法比较字符串。

Notice, the opposite direction doesn't have to hold. If the hashes are equal ($\text{hash}(s) = \text{hash}(t)$), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply $\text{hash}(s) = 0$ for each s. Now, this is just a stupid example, because this function will be completely useless, but it is a valid hash function. The reason why the opposite direction doesn't have to hold, is because there are exponentially many strings. If we only want this hash function to distinguish between all strings consisting of lowercase characters of length smaller than 15, then already the hash wouldn't fit into a 64-bit integer (e.g. unsigned long long) any more, because there are so many of them. And of course, we don't want to compare arbitrary long integers, because this will also have the complexity $O(n)$.

请注意，相反的方向不一定成立。如果哈希值相等 ($\text{hash}(s) = \text{hash}(t)$)，则字符串不一定必须相等。例如。对于每个 s ，有效的哈希函数将是简单的 $\text{hash}(s) = 0$ 。现在，这只是一个愚蠢的例子，因为这个函数将完全无用，但它是一个有效的哈希函数。相反方向不必成立的原因是因为字符串的数量呈指数级增长。如果我们只想让这个哈希函数区分所有由长度小于 15 的小写字符组成的字符串，那么哈希值就不再适合 64 位整数（例如 unsigned long long），因为它们太多了。当然，我们不想比较任意长整数，因为这也会有复杂度 $O(n)$ 。

So usually we want the hash function to map strings onto numbers of a fixed range $[0, m)$, then comparing strings is just a comparison of two integers with a fixed length. And of course, we want $\text{hash}(s) \neq \text{hash}(t)$ to be very likely if $s \neq t$.

所以通常我们希望哈希函数将字符串映射到固定范围的数字 $[0, m)$ ，那么比较字符串只是比较两个固定长度的整数。当然，如果 $s \neq t$ ，我们希望 $\text{hash}(s) \neq \text{hash}(t)$ 很可能出现。

That's the important part that you have to keep in mind. Using hashing will not be 100% deterministically correct, because two complete different strings might have the same hash (the hashes collide). However, in a wide majority of tasks, this can be safely ignored as the probability of the hashes of two different strings colliding is still very small. And we will discuss some techniques in this article how to keep the probability of collisions very low.

这是您必须牢记的重要部分。使用哈希不会 100% 确定性正确，因为两个完全不同的字符串可能具有相同的哈希（哈希冲突）。然而，在大多数任务中，可以安全地忽略这一点，因为两个不同字符串的哈希冲突的概率仍然很小。我们将在本文中讨论一些如何将碰撞概率保持在较低水平的技术。

Calculation of the hash of a string
计算字符串的哈希值

The good and widely used way to define the hash of a string s of length n is

定义长度为 n 的字符串 s 的哈希值的良好且广泛使用的方法是

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

where p and m are some chosen, positive numbers. It is called a **polynomial rolling hash function**.

其中 p 和 m 是一些选定的正数。它被称为多项式滚动哈希函数。

It is reasonable to make p a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of the English alphabet, $p = 31$ is a good choice. If the input may contain both uppercase and lowercase letters, then $p = 53$ is a possible choice. The code in this article will use $p = 31$.

将 p 设为大致等于输入字母表中字符数的素数是合理的。例如，如果输入仅由英文字母表中的小写字母组成，则 $p = 31$ 是一个不错的选择。如果输入可能同时包含大写和小写字母，则 $p = 53$ 是一个可能的选择。本文中的代码将使用 $p = 31$ 。

Obviously m should be a large number since the probability of two random strings colliding is about $\approx \frac{1}{m}$. Sometimes $m = 2^{64}$ is chosen, since then the integer overflows of 64-bit integers work exactly like the modulo operation. However, there exists a method, which generates colliding strings (which work independently from the choice of p). So in practice, $m = 2^{64}$ is not recommended. A good choice for m is some large prime number. The code in this article will just use $m = 10^9 + 9$. This is a large number, but still small enough so that we can perform multiplication of two values using 64-bit integers.

显然 m 应该是一个很大的数字，因为两个随机字符串碰撞的概率约为 $\approx \frac{1}{m}$ 。有时会选择 $m = 2^{64}$ ，因为此时 64 位整数的整数溢出的工作方式与模运算完全相同。但是，存在一种方法，可以生成冲突字符串（其工作独立于 p 的选择）。所以实际中不推荐使用 $m = 2^{64}$ 。 m 的一个不错的选择是一些大的素数。本文中的代码将仅使用 $m = 10^9 + 9$ 。这是一个很大的数字，但仍然足够小，以便我们可以使用 64 位整数执行两个值的乘法。

Here is an example of calculating the hash of a string s, which contains only lowercase letters. We convert each character of s to an integer. Here we use the conversion $a \to 1$, $b \to 2$, ..., $z \to 26$. Converting $a \to 0$ is not a good idea, because then the hashes of the strings a, aa, aaa, ... all evaluate to 0.

下面是计算字符串 s 的哈希值的示例，该字符串仅包含小写字母。我们将 s 的每个字符转换为整数。这里我们使用转换 $a \to 1$ 、 $b \to 2$ 、 ... 、 $z \to 26$ 。转换 $a \to 0$ 不是一个好主意，因为字符串 a 、 aa 、 aaa 、 ... 的哈希值全部计算为 0 。

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Precomputing the powers of p might give a performance boost.

预先计算 p 的幂可能会提高性能。

--------查找--------

2023年7月25日 8:56

KMP

2023年2月16日 19:15

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
void set_next_array(vector<int> &next, string &s)
{
    int i, j;
    if(next.empty() || s.empty())
    {
        return;
    }
    i = 0;
    j = -1;
    next[0] = -1;
    while(i < next.size() - 1)
    {
        if(j < 0 || s[i] == s[j])
        {
            i++;
            j++;
            next[i] = j;
        }
        else
        {
            j = next[j];
        }
    }
}
int  kmp_search(string &s, string &sub)
{
    int i = 0;
    int j = 0;
    if(s.empty() || sub.empty())
    {
        return -1;
    }
    vector<int> next(sub.size());
    set_next_array(next, sub);
    while(i < s.size() && j < (int)sub.size())
    {
        if(j < 0 || s[i] == sub[j])
        {
            i++;
            j++;
        }
        else
        {
            j = next[j];
```

```
        }
    }
    if(j == sub.size())
    {
        return i - j;
    }

    return -1;
}
int main()
{
    string s{"fskfjksldajflsdkajfklsajflsda"};
    string s1{"fs"};
    string s2{"sda"};
    cout << kmp_search(s, s1) << " " << s.find(s1) << endl;
    cout << kmp_search(s, s2) << " " << s.find(s2) <<endl;
}
```

1. 字符不匹配时，主串不需要回退，相等就向后滑动；子串回退步数取决于next数组
2. next数组的0位置元素为-1表示是为了方便实现子串从头开始匹配，并且主串索引要前进1，子串也前进1（变为0），这样就和匹配时操作一致（只要主串索引为-1，主串索引就加1，否则就不会前进）.
3. 所以在字符匹配实现中，子串索引下标i初始化为0，不是-1，因为主串要从第一个字符开始比较。
4. 若在T在某处匹配失败，对于匹配成功模式部分个数为q，如果有真前缀和真后缀相等的，且个数为k，那么从Ps+k（索引从1开始)重新开始比较（T位置不变）即可；

迷宫问题——广度优先搜索

2023年3月5日 13:08

描述

定义一个二维数组 N*M ，如 5 × 5 数组下所示：

```
int maze[5][5] = {
0, 1, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 0, 0,
0, 1, 1, 1, 0,
0, 0, 0, 1, 0,
};
```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的路线。入口点为[0,0],既第一格是可以走的路。

数据范围： $2 \leq n, m \leq 10$ ， 输入的内容只包含 $0 \leq val \leq 1$

输入描述：

输入两个整数，分别表示二维数组的行数，列数。再输入相应的数组，其中的1表示墙壁，0表示可以走的路。数据保证有唯一解,不考虑有多解的情况，即迷宫只有一条通道。

输出描述：

左上角到右下角的最短路径，格式如样例所示。

思路	将迷宫（二维数组每个元素为一个结点，然后根据有无墙壁（相邻结点是否右边）初始化这个图）用图表示，然后使用广度优先搜索找出最短路径（迪杰卡尔算法？——不是，笛卡尔是算带权最小路径）

示例1

```
输入：5 5
      0 1 0 0 0
      0 1 1 1 0
      0 0 0 0 0
      0 1 1 1 0
      0 0 0 1 0
输出：(0,0)
      (1,0)
      (2,0)
      (2,1)
      (2,2)
      (2,3)
      (2,4)
      (3,4)
      (4,4)
```

示例2

```
输入: 5 5
      0 1 0 0 0
      0 1 0 1 0
      0 0 0 0 1
      0 1 1 1 0
      0 0 0 0 0
输出: (0,0)
      (1,0)
      (2,0)
      (3,0)
      (4,0)
      (4,1)
      (4,2)
      (4,3)
      (4,4)

说明: 注意：不能斜着走！！
```

--------排序--------

2023年7月25日 8:59

topK

2023年2月28日 9:29

选择数据中前k大（或小）的数

一种期望O(N)的算法思路（只能期望，下限是nO(lg n)):(47条消息) 基于快速排序的O（N）时间复杂度的TopK算法原理_Paul-LangJun的博客-CSDN博客_topk算法 时间复杂度

其实，只要稍加修改快排算法，即可实现平均时间复杂度为O(N)的TopK算法，我们称之为QuickSelect(S, K)。

> 1、如果|S| = 1，返回S，否则若|S| < 20，则使用选择排序对S排序，选择最大的K个元素返回
> 2、选取一个枢纽元 v 属于 S。（**同快速排序步骤2**）
> 3、将集合 S - {v} 分割成 S1 和 S2。（**同快速排序步骤3**）
> 4、如果K <= |S1|，则K个元素必然全部位于集合S1中，并返回QuickSelect(S1, K)；如果K = |S1| + 1，则集合S1 与 枢纽元 v 恰好是所求的K个数，我们将 S1 和 v 一并返回；如果K > |S1| + 1，那么S1和枢纽元v必然是K个元素的一部分，剩余K - |S1| - 1 个元素必然存在集合 S2 中，因此我们应该返回 S1 + v + QuickSelect(S2, K - |S1| - 1)。

仔细分析该算法可得如下结论：

> - 快速选择算法的递归调用次数是快速排序的一半。
> - 两种算法的最坏情形都是O（N*N），即集合已排序的情况。
> - 算法平均时间复杂度为O（N）。

注	1. 这里第一点的20应该参数化（k），除非K永远不小于20

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
//这里考虑k不会很大，使用选择排序处理结果更快
void select_sort(vector<int> &data, int l, int r, int k)
{
    int max;
    int max_idx;
    max = data[l];
    max_idx = l;
    for(int i = l; i <= r; i++)
    {
        max = data[i];
        for(int j = i; j <= r; j++)
        {
            if(data[j] > max)
            {
                max = data[j];
                max_idx = j;
```

```
            }
        }
        cout << "topk:" << max << " ";
        swap(data[i], data[max_idx]);
        k--;
        if(k == 0)
        {
            cout << endl;
            return;
        }
    }
    cout << "error." << endl;
    return;
}
// 按快速排序原理将数据按中枢分开两半，左小右大
int split_half(vector<int> &data, int l, int r)
{
    int pivot, m;
    int i, j;
    pivot = data[r];
    for(i = l, j = l; j < r; j++)
    {
        if(data[j] <= pivot)
        {
            swap(data[i], data[j]);
            i++;
        }
    }
    swap(data[i], data[r]);
    return i;
}
void topk(vector<int> &data, int l, int r, int k)
{
    int m;
    if(l > r || k <= 0)
    {
        return;
    }
    //如果子集数量少于k，停止,topK就在该子集
    if((r-l+1) <= k)
    {
        //利用选择排序子集
        select_sort(data, l, r, k);
        return;
    }
    m = split_half(data, l, r);
    if((r-m) > k)
    {
        //只选择右半部部分子集
        topk(data, m+1, r, k);
    }
    else
    {
        //剩余右半子集不足k个，
        //那么就是右半子集、中枢
        topk(data, m, r, r-m+1);
        //和左半子集的一部分
        topk(data, l, m-1, k-(r-m+1));
    }

}
```

```
void print_topk(vector<int> &data, int k)
{
    if(data.empty() || data.size() < k)
    {
        cout << "incorrected parameters." <<endl;
        return;
    }
    topk(data, 0, data.size()-1, k);
}
int main()
{
    int k;
    vector<int> data1{0};
    vector<int> data2{0, 2, 1};
    vector<int> data3{0,32,999,32,88,11,-9,100,99,32};
    k=1;
    print_topk(data1, k);
    cout << "====================" << endl;
    print_topk(data2, k);
    cout << "====================" << endl;
    print_topk(data3, k);
    cout << "====================" << endl;
    k=2;
    print_topk(data1, k);
    cout << "====================" << endl;
    print_topk(data2, k);
    cout << "====================" << endl;
    print_topk(data3, k);
    cout << "====================" << endl;
    k=3;
    print_topk(data2, k);
    cout << "====================" << endl;
    print_topk(data3, k);
    cout << "====================" << endl;
    k=5;
    print_topk(data3, k);
}
```

--------动态规划--------

2023年7月25日 8:57

打家劫舍——动态规划

2022年9月4日 23:38

相关企业

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。**

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。

sum(0) = num[0]
sum(1)=max{num[0], num[1]}
sum(2)={num[2]+S(0), S(1)}
sum(i)={num[i] + S(i-2), S(i-1)}

购物清单——动态规划

2023年3月6日　　19:22

描述

王强决定把年终奖用于购物，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的，下表就是一些主件与附件的例子：

主件	附件
电脑	打印机，扫描仪
书柜	图书
书桌	台灯，文具
工作椅	无

如果要买归类为附件的物品，必须先买该附件所属的主件，且每件物品只能购买一次。
每个主件可以有 0 个、 1 个或 2 个附件。附件不再有从属于自己的附件。
王强查到了每件物品的价格（都是 10 元的整数倍），而他只有 N 元的预算。除此之外，他给每件物品规定了一个重要度，用整数 1 ~ 5 表示。他希望在花费不超过 N 元的前提下，使自己的满意度达到最大。
满意度是指所购买的每件物品的价格与重要度的乘积的总和，假设设第i件物品的价格为$v[i]$，重要度为$w[i]$，共选中了k件物品，编号依次为$j_1, j_2, ..., j_k$，则满意度为：$v[j_1] * w[j_1] + v[j_2] * w[j_2] + ... + v[j_k] * w[j_k]$。（其中 * 为乘号）
请你帮助王强计算可获得的最大的满意度。

1. 不能超总额N
2. 购买件数不能超m，每个物品只能买一次
3. 买附件必须买主件、每个主件会最多有两个附件
4. 物品价格是10的整数倍、物品重要度1~5
5. 满意度（物品价格*重要度之和）最大

Ma	
Mb	Ai
Mc	Aj、Ak

组合类型　　物件数　更正单位物品件的满意度

1. 买Ma、Mb、Mc　　==》1　==>y1
2. 买Ai+Mb　　==》2　==>y2
3. 买Aj+Mc　　==》2　==>y3
4. 买Ak+Mc　　==》2　==>y4
5. 买Aj+Ak+Mc　　==》3　==>y5

(买其中一类，下一次选择就要从选择集中删除这一类对应的物件
S ={yimaxj+yimax2+..+yimaxj <= N | i=1-5，N11+...+Nij <= m}
转换为分数背包贪心算法，但是处理起来复杂而且需要证明贪心成立——**不成立**；
应该当作0-1背包动态规划处理简单。

输入描述：

输入的第 1 行，为两个正整数N，m，用一个空格隔开：
（其中 N （ N<32000 ）表示总钱数， m （m <60 ）为可购买的物品的个数。）

从第 2 行到第 m+1 行，第 j 行给出了编号为 j-1 的物品的基本数据，每行有 3 个非负整数 v p q

（其中 v 表示该物品的价格（ v<10000 ）， p 表示该物品的重要度（ 1 ~ 5 ）， q 表示该物品是主件还是附件。如果 q=0 ，表示该物品为主件，如果 q>0 ，表示该物品为附件， q 是所属主件的编号）

输出描述：

输出一个正整数，为张强可以获得的最大的满意度。

示例1

输入：
```
1000 5
800 2 0
400 5 1
300 5 1
400 3 0
500 2 0
```
复制

输出：
```
2200
```
复制

示例2

输入：
```
50 5
20 3 5
20 3 5
10 3 0
10 2 0
10 1 0
```
复制

输出：
```
130
```
复

说明：由第1行可知总钱数N为50以及希望购买的物品个数m为5；
第2和第3行的q为5，说明它们都是编号为5的物品的附件；
第4~6行的q都为0，说明它们都是主件，它们的编号依次为3~5；
所以物品的价格与重要度乘积的总和的最大值为10*1+20*3+20*3=130

--------数据结构遍历--------

2023年7月25日 8:57

链表反转

2023年2月26日 22:16

1. 原地反转
   从尾部开始，一个个地反转指向，需要两个辅助指针
2. 头插法
   把链表头断开，再利用头插法，从头到尾地将结点一个个插进来，需要两个辅助指针
3. 迭代法
   类似原地反转，但是从头开始反转，需要三个辅助指针
4. 递归法
   其实就是原地反转，只不过以递归形式实现。

--------双指针--------

2023年7月25日 9:07

删除字符串空格

2023年7月25日 9:08

在C++中，字符串是可变的，因此可以更容易地实现时间复杂度为O(n)和空间复杂度为常量的算法来删除字符串中的所有空格。可以使用双指针技术，直接在原始字符串上进行修改，而无需额外的空间。

以下是C++示例代码实现该算法：

```cpp
#include <iostream>

void removeSpaces(std::string& s) {
    int writePtr = 0;

    for (int readPtr = 0; readPtr < s.length(); ++readPtr) {
        if (s[readPtr] != ' ') {
            s[writePtr] = s[readPtr];
            ++writePtr;
        }
    }

    s.resize(writePtr); // 更新字符串长度，去除末尾的空白字符
}

int main() {
    std::string inputStr = "  hello  world  ";
    removeSpaces(inputStr);
    std::cout << inputStr << std::endl; // 输出 "helloworld"

    return 0;
}
```

在这个示例代码中，我们直接修改了原始字符串 `s`，通过两个指针 `readPtr` 和 `writePtr` 来遍历和修改字符串，使得空间复杂度为常量。删除空格的操作是通过将非空格字符复制到合适的位置来实现的。在循环结束后，我们调用 `resize` 函数来更新字符串的长度，去除末尾的空白字符。

--------其他--------

2023年7月25日 8:58

密码合格验证程序——正则表达式

2023年3月7日 17:53

密码要求:
1.长度超过8位
2.包括大小写字母.数字.其它符号,以上四种至少三种
3.不能有长度大于2的包含公共元素的子串重复 （注：其他符号不含空格或换行）

数据范围：输入的字符串长度满足 $1 \leq n \leq 100$

输入描述：

一组字符串。

输出描述：

如果符合要求输出：OK，否则输出NG

示例1

输入：
```
021Abc9000
021Abc9Abc1
021ABC9000
021$bc9000
```
复制

输出：
```
OK
NG
NG
OK
```
复制

用正则表达式解决本问题是比较方便的。

2023年3月13日 15:23

给定一个 N 行 M 列的二维矩阵，矩阵中每个位置的数字取值为 0 或 1，矩阵示例如：

```
1 1 0 0
0 0 0 1
0 0 1 1
1 1 1 1
```

现需要将矩阵中所有的 1 进行反转为 0，规则如下：

1. 当点击一个 1 时，该 1 被反转为 0，同时相邻的上、下、左、右，以及左上、左下、右上、右下 8 个方向的 1 （如果存在 1）均会自动反转为 0；
2. 进一步地，一个位置上的 1 被反转为 0 时，与其相邻的 8 个方向的 1 （如果存在 1）均会自动反转为 0；

按照上述规则示例中的矩阵只最少需要点击 2 次后，所有均值 0 。请问，给定一个矩阵，最少需要点击几次后，所有数字均为 0?

输入

第一行输入两个整数，分别表示矩阵的行数 N 和列数 M，取值范围均为 [1,100]

接下来 N 行表示矩阵的初始值，每行均为 M 个数，取值范围 [0,1]

输出

输出一个整数，表示最少需要点击的次数

示例一

输入

```
3 3
1 0 1
0 1 0
1 0 1
```

输出

```
1
```

	1			1			1	
	1			1			1	

6个1，5*4*3*2*1=n!==由于感染式反转，所以实际可能比这更少==>6*(k1*k2*..ki),比如k1可能是5、4、3、2、1、0，k2对应可能是4、3、2、1、0（k1为4时）...

第一层：

循环查找原表，

找到1个后，计数器加1；创建一个新表，反转这个1；传递新表给第二层；cnt+=第二层的返回值

如果cnt《min，则min=cnt

返回min

第二层：同第一层的操作，“原表”即”新表“；

...

直到表中没有1；

代码

```
/*
给定一个 N 行 M 列的二维矩阵，矩阵中每个位置的数字取值为 0 或 1，矩阵示例如：
4 4
1 1 0 0
0 0 0 1
0 0 1 1
1 1 1 1
现需要将矩阵中所有的 1 进行反转为 0，规则如下：
1. 当点击一个 1 时，该 1 被反转为 0，同时相邻的上、下、左、右，以及左上、左下、右上、右下 8 个方向的 1 （如果存在 1）均会自动反转为 0；
2. 进一步地，一个位置上的 1 被反转为 0 时，与其相邻的 8 个方向的 1 （如果存在 1）均会自动反转为 0；
按照上述规则示例中的矩阵只最少需要点击 2 次后，所有均值 0 。请问，给定一个矩阵，最少需要点击几次后，所有数字均为 0?
*/
#include <iostream>
#include <vector>
using namespace std;
void clear_one(vector<vector<int>> &data, int i, int j)
{
    int rs, cs;

    if(data.empty())
    {
        cout << "error 2." << endl;
        return;
    }
    rs = data.size();
    cs = data[0].size();
    if(data[i][j] != 1)
    {
        return;
```

```
    }
    data[i][j] = 0;
    if(i-1 >= 0)
    {
        clear_one(data, i-1, j);
    }
    if(i+1 < rs)
    {
        clear_one(data, i+1, j);
    }
    if(j-1 >= 0)
    {
        clear_one(data, i, j-1);
    }
    if(j+1 < cs)
    {
        clear_one(data, i, j+1);
    }
    if(i-1 >= 0 && j-1 >= 0)
    {
        clear_one(data, i-1, j-1);
    }
    if(i-1 >= 0 && j+1 < cs)
    {
        clear_one(data, i-1, j+1);
    }
    if(i+1 < rs && j-1 >= 0)
    {
        clear_one(data, i+1, j-1);
    }
    if(i+1 < rs && j+1 < cs)
    {
        clear_one(data, i+1, j+1);
    }
}
int pop(vector<vector<int>> &data)
{
    int cnt = 0;
    int min = data.size();
    int rs, cs;
    bool flag = false;
    if(data.empty())
    {
        cout << "error." << endl;
        return -1;
    }
    rs = data.size();
    cs = data[0].size();
    for(int i = 0; i != rs; i++)
    {
        for(int j = 0; j != cs; j++)
        {
            if(data[i][j] == 1)
            {
                vector<vector<int>> new_data(data);
                clear_one(new_data, i, j);
```

```
                cnt = 1 + pop(new_data);
                flag = true;
                if(cnt < min)
                {
                    min = cnt;
                }
            }
        }
    }
    if(flag)
    {
        return min;
    }
    return 0;
}
int main()
{
    int row_size;
    int column_size;
    vector<vector<int>> data;
    cin >> row_size >> column_size;
    for(int i = 0; i != row_size; i++)
    {
        vector<int> tmp;
        for(int j = 0; j != column_size; j++)
        {
            int n;
            cin >> n;
            tmp.push_back(n);
        }
        data.push_back(tmp);
    }
    cout << "load data finished!." << endl;
    cout << pop(data) << endl;
}
```