《L4-CSP-J 复赛进阶算法与数据结构》

Day05 动态规划-Dynamic Programming

背包动态规划（上）

主讲人：小武老师

可达信奥_小武老师_keda.ac

动态规划-背包DP

动态规划其实是运筹学的一种最优化方法，是求解决策过程(decision process)最优化的数学方法。动态规划的应用极其广泛，包括工程技术、经济、工业生产、军事以及自动化控制等领域，并在背包问题、生产经营问题、资金管理问题、资源分配问题、最短路径问题和复杂系统可靠性问题等中取得了显著的效果。

可达信奥_小武老师_keda.ac

动态规划

动态规划（DP）不是某种具体算法，而是一种**思想**。

核心：把大问题转化为小问题，利用小问题的解推断出大问题的解。

动态规划

理查德·贝尔曼(Richard E. Bellman)

1957年出版《Dynamic Programming》
这是该领域的第一本著作。

研究多阶段决策过程(multistep decision process)的优化问题

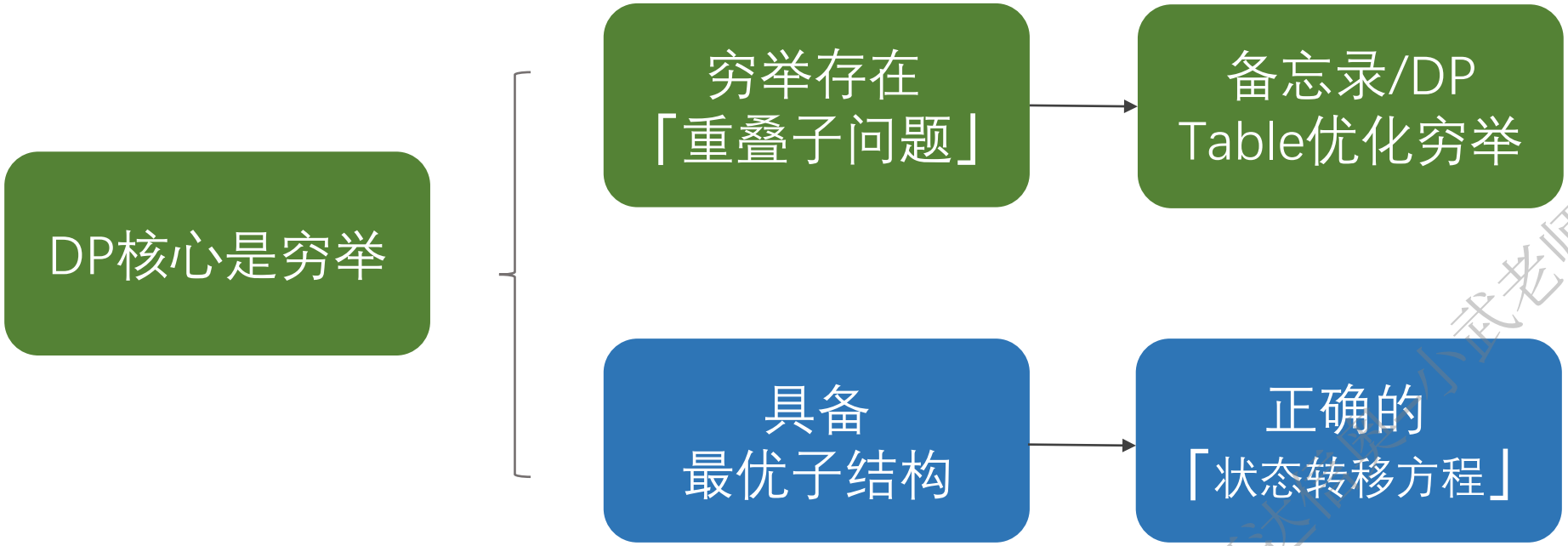
提出了著名的最优化原理(principle of optimality)把多阶段过程转化为一系列单阶段问题

利用各阶段之间的关系，逐个求解

创立了解决这类过程优化问题的新方法——动态规划。

动态规划问题的一般形式就是求最值

动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求**最长**递增子序列，**最小**编辑距离等等。

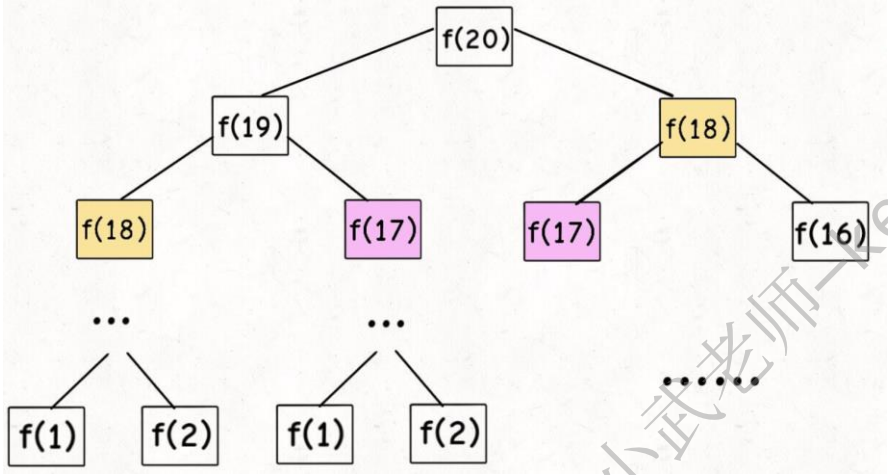


谈谈斐波那契数列

一、明白什么是重叠子问题

备注：（斐波那契数列没有求最值，所以严格来说不是动态规划问题）P0047 斐波那契数列 （递归超时）

```
int fib(int N) {
    if (N == 1 || N == 2)
        return 1;
    return fib(N - 1) + fib(N - 2);
}
```



递归算法的时间复杂度怎么计算？就是用子问题个数乘以解决一个子问题需要的时间。

$$O(2^n)$$

谈谈斐波那契数列

```cpp
#include<bits/stdc++.h>
using namespace std;
long long memo[10001];
int fib(int N) {
    if (N == 1 ) return 0;
    if (N == 2 ) return 1;
    return fib(N - 1) + fib(N - 2);
}
long long fib2(int n) {
    if (n == 1) return 0;
    if (n == 2) return 1;
    if (memo[n] != 0) return memo[n];
    memo[n] = fib2(n - 1) + fib2(n - 2);
    return memo[n];
}
```

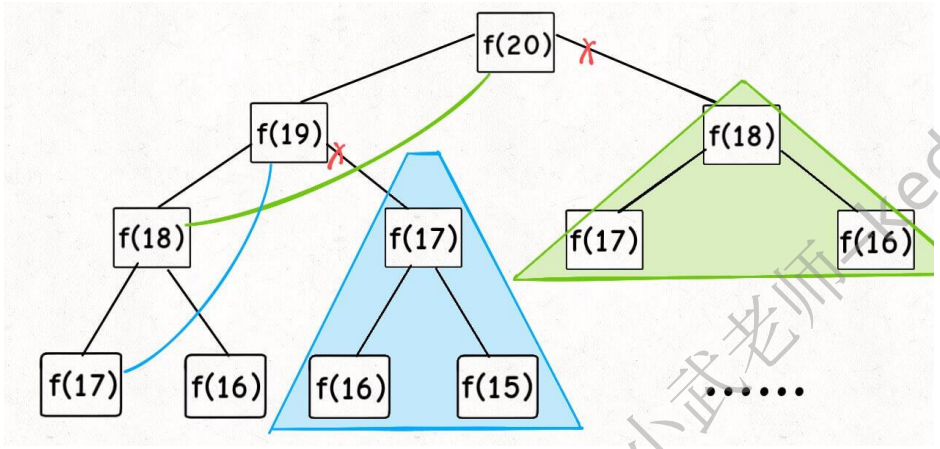
```cpp
int main(){
    int n;
    cin >> n;
    memset(memo,0,sizeof(0));
    cout << fib2(n);
    return 0;
}
```

fib2 解决了递归超时问题

二、带备忘录的递归解法

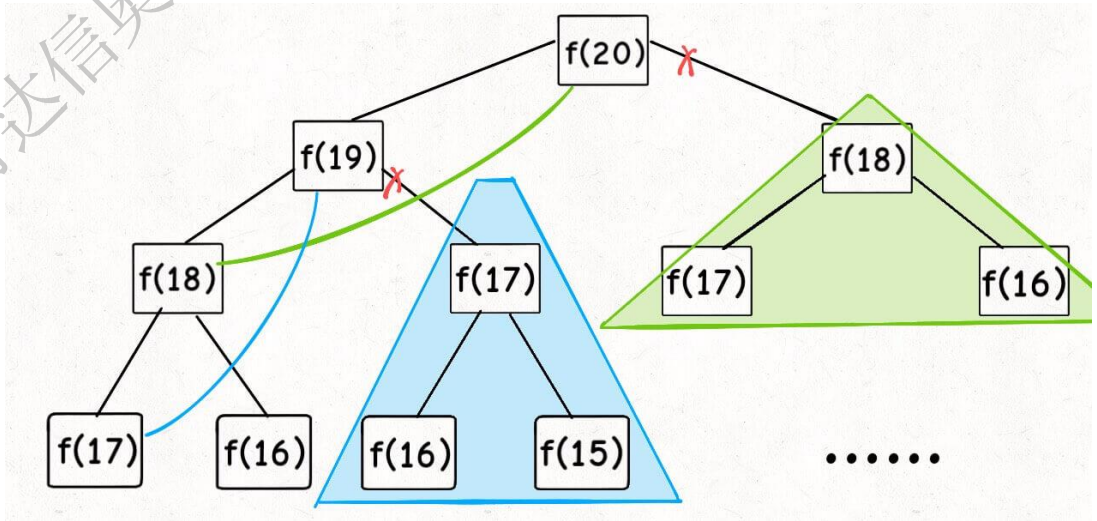
备注：（斐波那契数列没有求最值，所以严格来说不是动态规划问题）

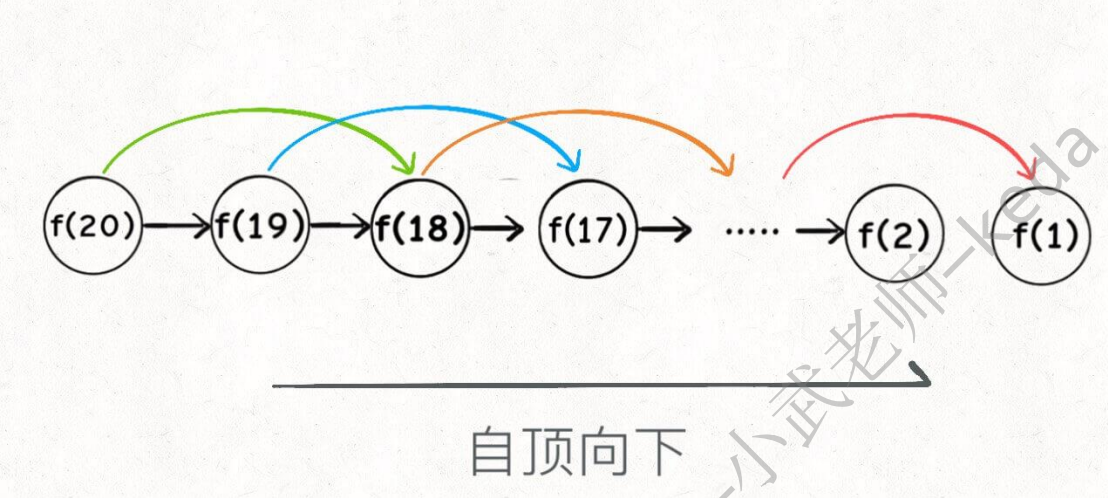
```
long long fib(int n) {
    if (n == 0 || n == 1) return n;
    if (memo[n] != 0) return memo[n];
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```



明白什么是重叠子问题

备注：（斐波那契数列没有求最值，所以严格来说不是动态规划问题）





本算法的时间复杂度是 O(n)，比起暴力算法，是降维打击

三、dp数组的迭代（递推）解法

如果将「备忘录」独立出来成为一张表，通常叫做 DP table，在这张表上完成「自底向上」的推算岂不美哉。

```
int fib(int N) {
    if (N == 0) return 0;
    // base case
    dp[0] = 0; dp[1] = 1;
    // 状态转移
    for (int i = 2; i <= N; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[N];
}
```

啥叫「自底向上」？反过来，我们直接从最底下、最简单、问题规模最小、已知结果的 f(1) 和 f(2)（base case）开始往上推，直到推到我们想要的答案 f(20)。这就是「递推」的思路，这也是动态规划一般都脱离了递归，而是由循环迭代完成计算的原因。

三、dp数组的迭代（递推）解法

$$f(n)=\begin{cases}1 & \text{if } n=1,2\\ f(n-1)+f(n-2) & \text{if } n>2\end{cases}$$

为啥叫「状态转移方程」？

其实就是为了听起来高端！！！

```
int fib(int n) {
    if (n == 0 || n == 1) {
        // base case
        return n;
    }
    // 分别代表 dp[i - 1] 和 dp[i - 2]
    int dp_i_1 = 1, dp_i_2 = 0;
    for (int i = 2; i <= n; i++) {
        // dp[i] = dp[i - 1] + dp[i - 2];
        int dp_i = dp_i_1 + dp_i_2;
        // 滚动更新
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i_1;
}
```

把空间复杂度从 O(n^2) 压缩到 O(n)

爬楼梯问题

今有n阶台阶。初始时站在0级，每次可以向上走1级或2级。问方案总数？ 暴力模拟：指数级复杂度

如果以$f[x]$表示“从0级走到x级的方案数”，

假设$f[1]$，$f[2]$...$f[n-1]$全都已知，如何利用这些信息推出$f[n]$？

走到$f[n]$，要么是从$n-1$级走上来的，要么是从$n-2$级来的。

依据加法原理

$$f[n] = f[n-1] + f[n-2]$$

凑零钱问题

今天你手上有无限的面值为1,5,11 元的硬币。

给定n，问：至少用多少枚硬币，可以恰好凑出n元？

依据生活经验，我们可以采用这种策略：

先尽量用11的，然后尽量用5元的……以此类推。

贪心？

凑零钱问题

今天你手上有无限的面值为1,5,11 元的硬币。

给定n，问：至少用多少枚硬币，可以恰好凑出n元？

假如

n = 15 时答案是 3，构造方法为 5+5+5

n = 12 时答案是 2，构造方法为 11+1

如果我们要凑出15，贪心策略是：

15 = 11+4*1,共用5枚硬币。

而最佳策略是：

15 = 3*5,共用3枚硬币。

凑零钱问题

那该怎么避免贪心的“鼠目寸光”呢？

强行枚举使用的硬币？复杂度太高。

- 观察一波性质？

凑零钱问题

我们记“凑出n需要用到的最少硬币数量”为 $f(n)$

w=15时

我们取了 11，接下来面对 w=4 的情况。则cost = f(4) + 1

如果我们取 5，接下来就面对 w=10 的情况。则cost = f(10) + 1

那么，w=15时，我们选哪枚硬币呢？ cost最低的那一个!

11： $\text{cost} = f(4) + 1 = 4 + 1 = 5$

5： $\text{cost} = f(10) + 1 = 2 + 1 = 3$

1： $\text{cost} = f(14) + 1 = 4 + 1 = 5$

选择5，$f(15) = 3$，即为答案!

P0461. 硬币问题

这是一个很棒的性质：

$f(n)$ 只与 $f(n-1), f(n-5), f(n-11)$ 相关

更确切地说：

$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$

- $f(n)$只与$f(n-1), f(n-5), f(n-11)$相关。
- 我们只关心$f(w)$的值，不关心是怎么取到w的。

凑零钱问题

P0461. 硬币问题 $O(n)$

```cpp
#include<bits/stdc++.h>
using namespace std;
int dp[100001];
const int INF = 0x3f3f3f3f;
int main(){
    int n,cost;
    cin >> n;
    dp[0] = 0;
    for(int i = 1; i <= n; i++){
        cost = INF;
        if(i-1 >= 0) cost = min(cost, dp[i-1]+1);
        if(i-5 >= 0) cost = min(cost, dp[i-5]+1);
        if(i-11 >= 0) cost = min(cost, dp[i-11]+1);
        dp[i] = cost;
    }
    cout << dp[n];
    return 0;
}
```

$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$

P0461. 硬币问题

这个算法的时间复杂度显然是$O(n)$.为什么比暴力要快呢?

我们的暴力枚举了“使用的硬币”，然而这属于冗余信息。

我们要的是答案，根本不关心这个答案是怎么凑出来的。

要求出 $f(15)$，只需要知道 $f(14), f(10), f(4)$ 的值。其他信息不需要。舍弃了冗余信息。

求出$f(n)$，只需要知道几个更小的$f(c)$.

我们将求解$f(c)$称作求解$f(n)$的“子问题”。

理解什么是状态

我们用**“大事化小，小事化了”**的思想，解决了上楼梯问题和硬币问题。大事能转化成小事，是因为大事和小事都有一样的**形式**：

爬楼梯问题

大问题：爬上n级有多少种方案

小问题：爬上$n-1$级有多少种方案、爬上$n-2$级有多少种方案

它们都是**“爬上$\times\times$级有多少种方案”**这一类问题。

硬币问题

大问题：凑出n元钱的最少硬币数

小问题：凑出n－1, n－5, n－11元的最少硬币数

它们都是**“凑出$\times\times$元钱所需最少硬币数”**这一类问题。

理解什么是状态

只有大问题和小问题拥有**相同的形式**，才能考虑大事化小。如果满足这个要求，那么我们遇到的每个问题，都可以很简洁地表达。我们把可能遇到的每种**“局面”**称为状态。

硬币问题中，要表达“我们需要凑出n 元钱”这个局面，可以设计状态：

“$f[x]$ 表示凑出x 元用的最少硬币数”。

上楼梯问题中，设计状态：

“$f[x]$ 表示走上x 级的方案数”。

设计完状态之后，只要能**利用小状态的解求出大状态的解**

凑零钱问题

具备
最优子结构 → 正确的
「状态转移方程」

- 回顾我们对$f(n)$的定义：
- 我们记“凑出n需要用到的最少硬币数量”为$f(n)$.

- $f(n)$的定义就已经蕴含了“最优”。
- 利用w=14,10,4的最优解，我们即可算出w=15的最优解。

- 大问题的最优解可以由小问题的最优解推出，这个性质叫做“最优子结构性质”。

无后效性

- 一旦$f(n)$确定，“我们如何凑出$f(n)$”就再也用不着了。
- 要求出$f(15)$，只需要知道$f(14), f(10), f(4)$的值，而$f(14), f(10), f(4)$是如何算出来的，对之后的问题没有影响。
- “未来与过去无关”，这就是无后效性。
- （其严格定义：如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响。）

凑零钱问题

什么情况下我们能使用DP呢？

- 我们能将大问题拆成几个小问题，且满足
  - 无后效性
  - 最优子结构性质

凑零钱问题

什么情况下我们能使用DP呢？

- 我们能将大问题拆成几个小问题，且满足
  - 无后效性
  - 最优子结构性质

关于DP的本质

1、DP为什么会快？

2、DP的核心是什么？

3、关于终极的思考？我是谁？我从哪里来？我要到哪里去？

问题一：DP为什么会快？ DP自带剪枝

无论是DP还是暴力，我们的算法都是在可能解空间内，寻找最优解。

来看硬币问题：

暴力做法是枚举所有的可能解，这是最大的可能解空间。

DP是枚举有希望成为答案的解。这个空间比暴力的小得多。

DP舍弃了一大堆不可能成为最优解的答案。譬如硬币问题：

15 = 5+5+5 被考虑了。

15 = 5+5+1+1+1+1+1 从来没有考虑过，因为这不可能成为最优解。

问题二：DP的核心思想？

大事化小，小事化了。

DP的核心思想：

尽量缩小可能解的空间

在暴力算法中，可能解空间往往是指数级的大小；如果我们采用DP，那么有可能把解空间的大小降到多项式级。

一般来说，解空间越小，寻找解就越快。这样就完成了优化

↓ DP的操作过程

将一个大问题转化成几个小问题；

求解小问题；

推出大问题的解。

问题三：如何设计DP算法?

DP的设计步骤：

我是谁？ ——设计状态，表示局面 x

我从哪里来？ f(x)

我要到哪里去？ ——设计转移

一般而言，DP的难点，在初学时是如何设计状态；在学习深入一些之后，变成了如何设计转移；在省选/NOI级别，又变成了如何设计状态。

DP总结

1、计算机解决问题其实没有任何奇技淫巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考“如何穷举”，然后再追求“如何聪明地穷举”。

2、列出状态转移方程，就是在解决“如何穷举”的问题。之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不那么容易穷举完整。

3、备忘录、DP table 就是在追求“如何聪明地穷举”。用空间换时间的思路，是降低时间复杂度的不二法门。

4、DP 设计三板斧：我是谁？我从哪里来？（设计状态，表示局面f(x)）我要到哪里去？

例题选讲

P0449. 最长上升子序列　　P0464 最大子段和

P0449 最长上升子序列

设计状态

以$f[x]$表示“以$a[x]$结尾的上升子序列，最长有多长”！

那么，答案就是$f[1]$，$f[2]...f[n]$里面的最大值。

问题来了，如何求出f数组？提示：思考$f[x]$**从哪里来**。

$f[x]$表达的是“以$a[x]$结尾的最长的上升子序列长度”。这个最长的子序列，一定是把$a[x]$接在某个上升子序列尾部形成的！

P0449 最长上升子序列

$f[x]$ 表达的是“以$a[x]$ 结尾的最长的上升子序列长度”。这个最长的子序列，一定是把$a[x]$ 接在某个上升子序列尾部形成的！

数组a:[1,3,4,7,2,6,8,5]

考虑f[8]，它的来源有：

自己一个元素作为一个序列。长度为1.

接在a[1] 后面。长度为f[1]+1=2

接在a[2] 后面。长度为f[2]+1=3

接在a[3] 后面。长度为f[3]+1=4

接在a[5] 后面。长度为f[5]+1=3

$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p]+1\}$$

P0449 最长上升子序列

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[10001];
int dp[10001];
int main(){
    int n, maxn = 0;
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++){
        dp[i] = 1;
        for(int j = 1; j < i; j++){
            if(a[j] < a[i]) dp[i] = max(dp[j]+1, dp[i]);
        }
        maxn = max(maxn, dp[i]);
    }
    cout << maxn;
    return 0;
}
```

$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p] + 1\}$$

P0464 最大子段和

P0464 给出一个整数序列，问最大子段和。e.g. [2,-1,3,-5,3]的最大子段和是2-1+3=4. 要求一个$O(n)$的算法。

- 如何设计状态?
- dp[i]表示[1,i]位的最大子段和。

- 状态i从哪里来?
- 要么合并上之前的最大子段和(dp[i-1])，要么另起炉灶。

- dp[i]=max(dp[i-1]+a[i],a[i])

P0464 最大子段和

P0464 给出一个整数序列，问最大子段和。e.g. [2,-1,3,-5,3]的最大子段和是2-1+3=4. 要求一个$O(n)$的算法。

- 如何设计状态?
- dp[i]表示[1,i]位的最大子段和。

- 状态i从哪里来?
- 要么合并上之前的最大子段和(dp[i-1])，要么另起炉灶。

- dp[i]=max(dp[i-1]+a[i],a[i])

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[10001];
int dp[10001];
int main(){
    int n, maxn = 0x80000000;
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    dp[1] = a[1];
    for(int i = 2; i <= n; i++){
        dp[i] = max(dp[i-1]+a[i], a[i]);
        maxn = max(maxn, dp[i]);
    }
    cout << maxn;
    return 0;
}
```

课后习题与实验

Talk is cheap, show me the code !

下节课见啦！