STUACM 第二次集训

2019/10/25

- 快速幂
  - 引子，求$S=11^{128}$
    - 直接循环128次求积

      ```
      int s = 1;    // 11^128肯定会爆int的，这里仅仅是为了演示
      int a = 11;
      int n = 128;
      for(int i=1;i<=n;++i) s *= a;
      ```

      需要乘法次数为**128**次，没有更快的方法了吗？
    - 注意到$128=2*64$，有$S=11^{128}=(11^{64})^2$

      ```
      int s = 1;    // 11^128肯定会爆int的，这里仅仅是为了演示
      int a = 11;
      int n = 64;
      for(int i=1;i<=n;++i) s *= a;
      s *= s;
      ```

      现在需要的乘法次数为**64+1**次
    - 同样的，$64=2*32$，$32=2*16$，……，有$S=((((((11^2)^2)^2)^2)^2)^2)^2$

      ```
      int s = 1;    // 11^128肯定会爆int的，这里仅仅是为了演示
      int a = 11;
      int n = 7;
      for(int i=1;i<=n;++i) a *= a;
      s = a;
      ```

      现在需要的乘法次数仅为**7**次，从$O(n)$降到了$O(\log_2 n)$！
  - 128恰好是2的幂，那求$S=11^{73}$呢？
    - 可以把73拆成多个2的幂的和啊！
    - $73=64+8+1=2^6+2^3+2^0$，有
      $S=11^{64}*11^8*11^1=S=(((((11^2)^2)^2)^2)^2)^2*((11^2)^2)^2*11$
    - 上述计算需要的乘法次数就是**6+3+0+1+1**次吗？
    - 在求11^{2^6}时，不是求过11^{2^3}和11^{2^0}了嘛！

- ```
  int s = 1;     // 11^73肯定会爆int的，这里仅仅是为了演示
  int a = 11;
  int ap2[7];     // 记录a^(2^i)
  ap2[0] = a;
  int n = 6;
  for(int i=1;i<=6;++i){
      a *= a;
      ap2[i] = a;
  }
  s = ap2[6] * ap2[3] * ap2[0];
  ```

  现在需要乘法次数为**6+3**次**

- 可以把73拆成多个2的幂的和？根据73的二进制来拆

  $(73)_{10} = (100101)_2$

  ```
  int pow(int a, int n){
      int res = 1;
      while(n){
          if(n&1){
              res *= a;
          }
          a *= a;
          n >>= 1;
      }
      return res;
  }

  int pow_mod(int a, int n, int mod){     // 加上取模的版本
      int res = 1;
      while(n){
          if(n&1){
              res *= a;
              res %= mod;
          }
          a *= a;
          a %= mod;
          n >>= 1;
      }
      return res;
  }
  ```

- 参考资料

* 初窥动态规划
  - 引子
    - 题目：有一座高度是10级台阶的楼梯，从下往上走，每跨一步只能向上1级或者2级台阶。求出一共有多少种走法。
    - 先定义$f[i] := \text{到达第}i\text{层楼梯的走法数}$
    - 从上往下，从第10层往下，画一棵树
    - 再从下往上看这棵树，发现：
      - 只要结点代表的是第i层，那么$f[i]$总是相等的
      - $j > i$，那么求$f[j]$与$f[i]$有关，而求$f[i]$与$f[j]$一定无关
      - 把结点的值用$f[i]$代替，向上一个个简化
      - 得到递推式：$f[i] = f[i-1] + f[i-2]$
  ```

- 实现

```
int f[11];
f[0] = 0;
f[1] = 1;
f[2] = 2;
for(int i=3;i<=10;++i) f[i] = f[i-1] + f[i-2];
```

- 总结步骤：
  - 定义状态
  - 初始化基本状态
  - 得出递推式（状态转移方程）
  - 按递推方向求解

- **01背包——动态规划中的一类题**
  - 适用问题：有n个物品，它们有各自的体积和价值，现有给定容量的背包，如何让背包里装入的物品具有最大的价值总和？

i（物品编号）	1	2	3	4
w（体积）	2	3	4	5
v（价值）	3	4	5	6

  - 例题：Bone Collector
    - 对该题样例的分析可看01背包样例演示，看完PPT再看代码效果更佳哦！
    - **定义状态：**
      $max\_Value[i][j] :=$ 用容量为j的背包，考虑前i个物品的取舍，所能取得的最大价值和
    - **初始化基本状态：**
      - 有0个物品可以取时：$max\_Value[0][j] = 0$
      - 背包容量为0时：$max\_Value[i][0] = 0$
    - **得出递推式：**
      - **思路：** 背包容量为j，考虑第$i+1$个物品时，状态为$(i+1, j)$。如果取了这件物品，那么问题就变成了：背包容量为$j-$(物品i的重量)时，从前i个物品做选择，能够取得最大价值和是？则从$(i+1, j)$状态转移到$(i, j-$(物品i的重量)$)$；如果不取这件物品，那完全可以当这件物品不存在，那么问题就变成了：背包容量为j时，从前i个物品做选择，能够取得最大价值和是？则从$(i+1, j)$状态转移到(i, j)。两种做法选择能获取最大价值的那种。
      - **递推式：**
        $max\_Value[i][j] = max(max\_Value[i-1][j-weight[i]] + value[i], max\_Value[i-1][j])$
    - **按递推方向求解：**
      - 一个一个物品*加进去*考虑，i是递增的；对于固定的i，j遍历所有可能的背包容量，j可以是任意顺序的。

```
#include <bits/stdc++.h>
using namespace std;
const int mx = 1002;     // 最多多少物品。依题目数据，该值同时也是最大容量
int max_Value[mx][mx] = {0};
int v[mx]; // value
int w[mx]; // weight
int main()
{
    int T, n, v;     // T组数据，n个物品，背包容量最大为v
    scanf("%d", &T);
    while (T--)
    {
```

```
        scanf("%d %d", &n, &v);
        for (int i = 1; i <= n; ++i)
            scanf("%d", &v[i]);
        for (int i = 1; i <= n; ++i)
            scanf("%d", &w[i]);
        memset(max_value, 0, sizeof(max_value));    // 用于把整个数组
清0

        for (int i = 1; i <= n; ++i){    // 逐个物品加进去考虑
            for (int j = 0; j <= v; ++j){    // 需要在前i个物品做选择
时，任意背包容量所对应的最大价值
                if (j >= w[i]){
                    max_value[i][j] = max(max_value[i - 1][j -
w[i]] + v[i], max_value[i - 1][j]);
                }else{
                    max_value[i][j] = max_value[i - 1][j];
                }
            }
        }

              /*
              for (int i = 1; i <= n; ++i){
                  for (int j = v; j >= 0; --j){
                      if (j >= w[i]){
                          max_value[i][j] = max(max_value[i - 1][j
- w[i]] + v[i], max_value[i - 1][j]);
                      }else{
                          max_value[i][j] = max_value[i - 1][j];
                      }
                  }
              }
              */

         printf("%d\n", max_value[n][v]);
    }
    return 0;
}
```