题型

- **填空** 20分 10个
- **选择题** 10分 5个
- **简答题** 30分 3题
- **问答题** 40分 4题

填空

1.回溯法、分支限界一般用于求解什么样的问题?

- 随着实例规模的增长，问题的选择次数至少呈指数增长
- 回溯法——dfs (深度优先搜索)
  - n皇后问题
  - 哈密顿回路问题
  - 子集和问题
- 分支界限法——bfs (广度优先搜索)
  - 分配问题
  - 背包问题
  - 旅行商问题

2.Kruskal、Prim基于什么策略?

- 基于最小生成树的MST性质的**贪心**策略

3.二分复杂度

- 最好情况：一次命中——$O(1)$
- 最坏情况：只剩最后一个——$O(log_2 n)$
- 平均：$O(log_2 n)$

4.分治法实现快排

- 最好：$O(nlogn)$
- 最坏：$O(n^2)$
- 平均：$O(nlogn)$

5.回溯、分支界限对于树的遍历

- 回溯——对应树的**前序遍历**、**中序遍历**、**后序遍历**
- 分支界限——对应树的**层次遍历**
- 适用场景：(待补)

6.01背包求解策略?

- 蛮力——穷举查找：枚举 n 个物品放与不放两种状态，时间复杂度 $O(2^n)$
- **动态规划**：设有 n 个物品，w 的背包容量，则时间和空间复杂度都是 $O(nw)$
- 回溯法(需要排序)
- 分支界限法(需要排序)

7.动态规划一般求解什么样的问题？ 遵循什么样的法则?

- 求解**具有某种最优性质**的问题，即求解**最优化**问题
- 遵循**最优化法则**

8.最近点对和凸包问题基于什么策略?

- **分治法**
- 蛮力法

9.递归算法定义？ 递归算法是什么什么样的、什么什么样的是递归算法

- **递归算法**在计算机科学中是指**一种通过重复将问题分解为同类的子问题而解决问题的方法。**

10.快包法的时间复杂度?

- 最优：$O(nlog_2n)$
- 最坏：$O(n^2)$
- 一般采用**分治法**的策略

11.2-3树

- 满2-3树的高度 h : $log_3(n+1)-1 \leq h \leq log_2(n+1)-1$
- 应用：查找

选择

Dijkstra

- 基于贪心策略

渐近符号性质

- 书本P_{41-43}

贪心算法和动态规划是否可以解决问题的相关性

- 贪心算法和动态规划都要满足**最优子结构**

贪心算法和分支限界解决什么问题

- 若是两种策略都能解决的问题：单源最短路径
- 贪心解决最优问题，需要每一步选择满足可行、局部最优和不可取消原则。
- 分支界限解决随着实例规模的增长，问题的选择次数至少呈指数增长的问题。

变治算法解决哪些问题

- LU分解(高斯消去)
- AVL树
- $2-3$树

用动态规划解决的问题

- 背包问题
- 单源最短路径
- 最优二叉查找

简答题

相容问题

- //------问题：有n项活动申请使用同一个礼堂，每项活动有一个开始时间和一个截止时间。如果任何两个活动不能同时举行，问如何选择这些活动，从而使得被安排的活动数量达到最多。
- 按照开始时间排序，会发现得不到最优解

- 按照活动时间长度排序，也无法得到最优解

- 所以按照结束时间排序
  - 列一个例子

实例：**截止时间已经排序**

I	1	2	3	4	5	6	7	8	9	10
s_i	1	3(4>3:与活动1不相容	2(4>2:与活动1不相容	5(5>4:与活动1相容)	4	5	6	8	8	2
f_i	**4(活动1)**	5	6	**7**	9	9	10	**11**	12	13

- 时间复杂度：对截止时间的排序

Kruskal、Prim算法的比较

- 区别
  - 在适用性上，$Prim$ 适用于稠密图，即顶点相对较少边数相对较多
  - $Kruskal$ 适用于稀疏图，即边数相对较少
- 联系：两者都是基于贪心策略来求最小生成树的算法
- 实现思路
  - $Prim$ 是根据目前已经生成的生成树当中，添加距离树中各个顶点最近的顶点，重复此操作
    - 进行堆优化的时间复杂度 $O(elogn)$,e 是边数, n 是顶点数
    - 没有优化的是 $O(n^2)$
  - $Kruskal$ 是根据所有边进行升序排序，若边上两顶点都不在同一棵子树上，便用该边将两棵子树合并
    - 时间复杂度 $O(eloge)$,e 是边数

- 优点
  - $Prim$ 在连通图足够密集时,可显著的提高运行速度
  - $Kruskal$ 算法的时间复杂度主要由对边权值的排序决定，所以在边数较少的时候，使用该算法构造的最小生成树效果较好

AVL树

【数据结构】AVL树（平衡二叉树）画法 速成教学 - 知乎 (zhihu.com)

主定理及其应用

- 适用于某种特定的递归式
  - $T(n)=aT(\frac{n}{b})+f(n),$其中$a\geq 1,b>1$
  - $f(n)$ 是一个渐近趋正的函数，即 $f(n)>0,n\geq n_0$

主定理

定理：设$a \geq 1, b>1$为常数，$f(n)$为函数，$T(n)$为非负整数，且$T(n)=aT(n/b)+f(n)$，则

1. 若$f(n)=O(n^{\log_b a-\varepsilon}), \varepsilon>0$，那么
$$T(n)=\Theta(n^{\log_b a})$$
存在ε

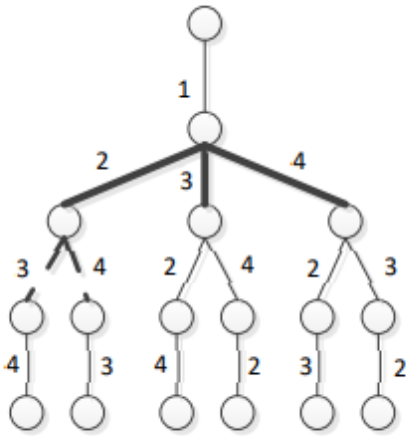
2. 若$f(n)=\Theta(n^{\log_b a})$，那么
$$T(n)=\Theta(n^{\log_b a}\log n)$$
存在ε

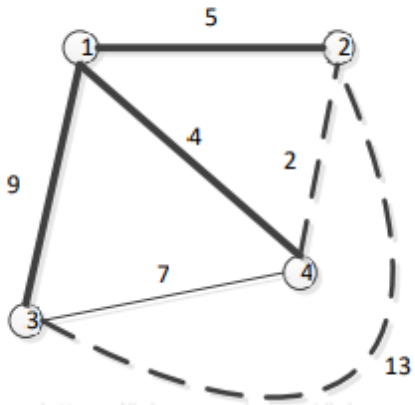
3. 若$f(n)=\Omega(n^{\log_b a+\varepsilon}), \varepsilon>0$，且对于某个常数$c<1$和充分大的$n$有$af(n/b) \leq cf(n)$，那么
$$T(n)=\Theta(f(n))$$
存在c和n_0

- 应用
  - 二分搜索：每次问题规模减半→$a=1, b=2, \varepsilon=0$,可得第二类：$T(n)=\Theta(logn)$
  - 快速排序：$a=2, b=2, \varepsilon=0$，可得第二类：$T(n)=\Theta(nlogn)$

货郎问题

- 问题：某售货员要到若 干城市去推销商品，各城市之间的距离为已知。他要选定一条驻地出发所有城市最后回到驻地的旅游路线，使得总的路程最短。
- 数学模型：已知一个带权完全图，求权值和最短的一条哈密尔顿回路。
- 设计：建立搜索树





- 复杂度分析：
  - 最坏情况下：时间复杂度$O(n!)$
  - 但在实际中，可以削减不必要的搜索分支，来优化时间复杂度

Floyd

- 画图过程题

图 8.16 告诉我们如何对图 8.14 中的图应用 Floyd 算法。

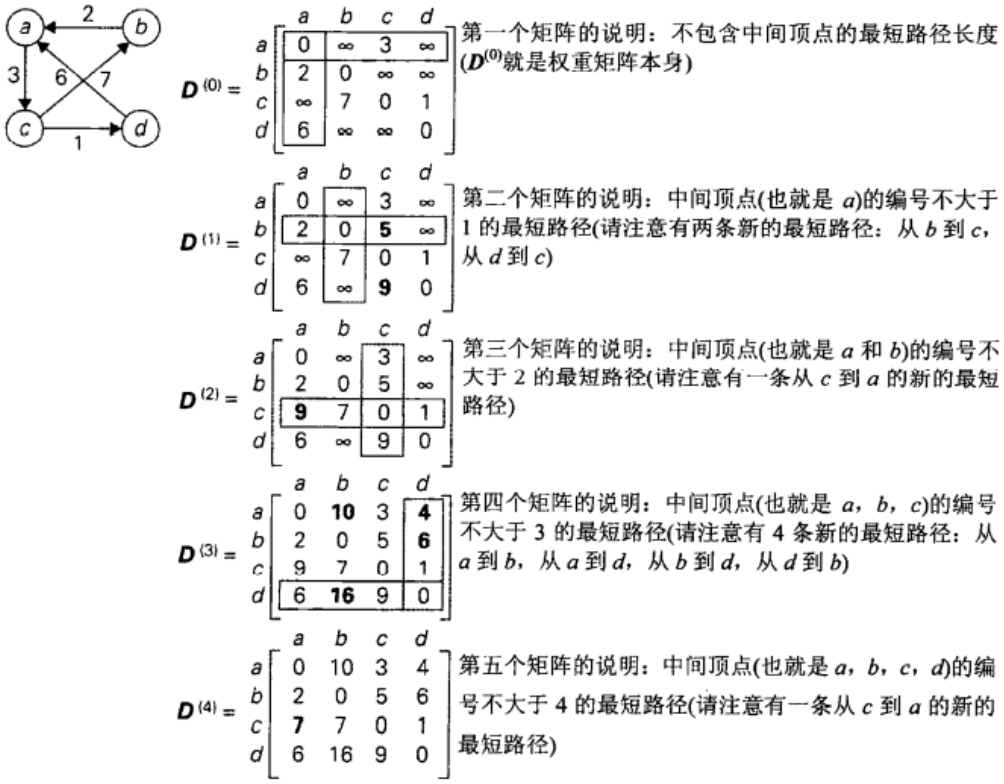


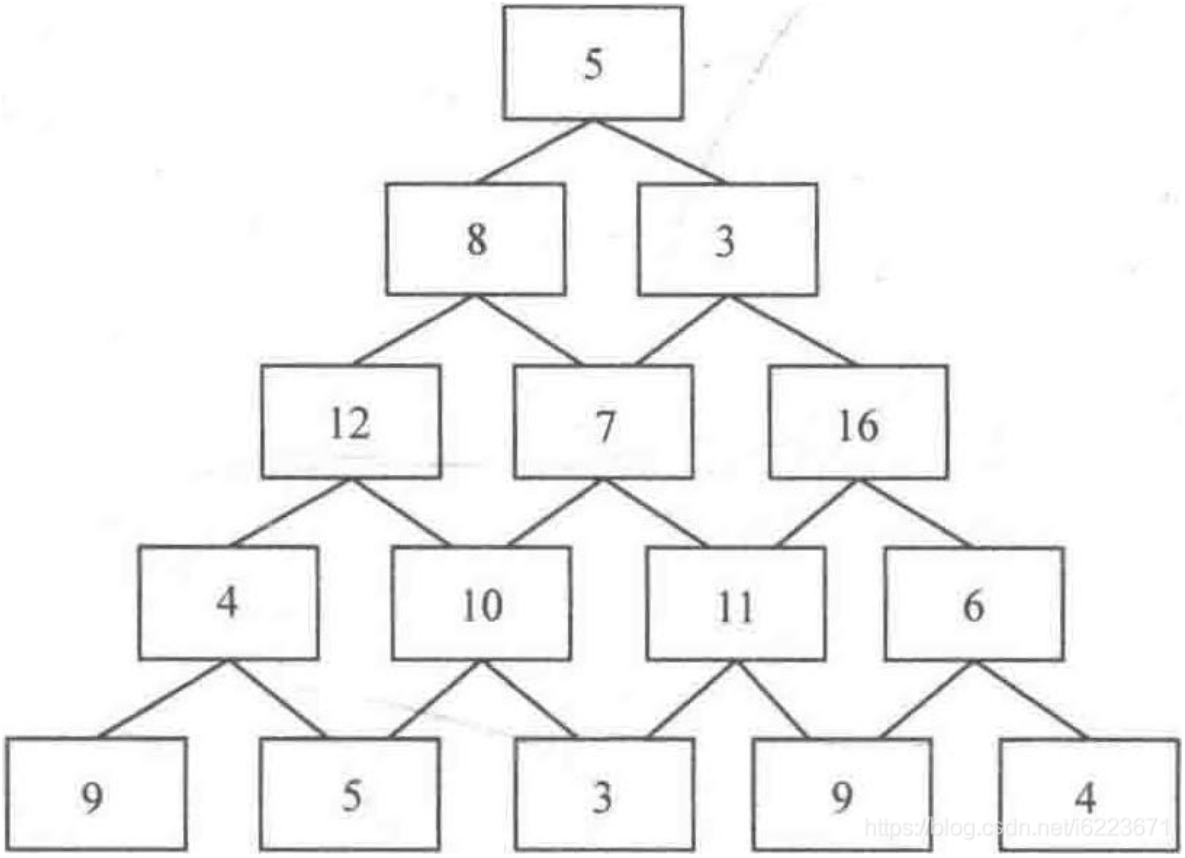
图 8.16　对给出的图应用 Floyd 算法。被改写的元素用粗体字表示

以下是 Floyd 算法的伪代码。它利用了这样一个事实，即序列(8.12)中的下一个矩阵可以在它的前趋矩阵上进行改写。

问答题

数塔问题

- 题目：一些数字排成数塔的形状，其中第一层有一个数字，第二层有两个数字... 第n 层有n 个数字。现在要从第一层走到第n 层，每次只能走向下一层连接的两个数字中的一个，问：最后将路径上所有数字相加后得到的和最大是多少？



- 设计：$f[n][m]$ 表示第 n 层第 m 个塔的权值，$dp[n][m]$ 表示第 n 层第 m 个塔为起点最底层的路径最大值
- 最终答案为 $dp[1][1]$

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int f[N][N], dp[N][N];
int main() {
    int n; cin >> n;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= i; ++j)
            cin >> f[i][j];
    for (int i = 1; i <= n; ++i) dp[n][i] = f[n][i];//----初始化最底层的dp值
    for (int i = n - 1; i >= 1; --i)
        for (int j = 1; j <= i; ++j)
            dp[i][j] = max(dp[i + 1][j], dp[i + 1][j + 1]) + f[i][j];
    cout << dp[1][1] << endl;
}
```

查找一个队列多数元素 分治策略

-
-

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int a[N];
int solve(int l, int r) {
```

```
    if (l == r) return a[l];
    int mid = (l + r) / 2;
    int left_num = solve(l, mid);
    int right_num = solve(mid + 1, r);
    if (left_num == right_num) return left_num;
    //----判断在[l,r]这个区间里，left_num和right_num哪个次数比较多
    int left_num_cnt = 0, right_num_cnt = 0;
    for (int i = l; i <= r; ++i) {
        if (left_num == a[i]) ++left_num_cnt;
        if (right_num == a[i]) ++right_num_cnt;
    }
    if (left_num_cnt > right_num_cnt) return left_num;
    else return right_num;
}
int main() {
    int n; cin >> n;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    cout << solve(1, n) << '\n';
}
```

二分查找 分治策略

- 假设在一个有序数组 $a[N]$ 中查找第一个大于 k 的数
- ```
  #include <bits/stdc++.h>
  using namespace std;
  const int N = 1e2 + 10;
  int a[N];
  int main() {
      int n, k; cin >> n >> k;
      for (int i = 0; i < n; ++i) cin >> a[i];//----假定已经有序
      int l = 0, r = n - 1, ans = a[0];
      while (l <= r) {
          int mid = (l + r) / 2;
          if (a[mid] > k) {
              ans = a[mid];
              r = mid - 1;
          }
          else l = mid + 1;
      }
      cout << ans << '\n';
  }
  ```

回溯策略 回文

- 将一个字符串分割成多个回文串
- ```
  #include <bits/stdc++.h>
  using namespace std;
  const int N = 1e2 + 10;
  ```

```
bool check(string s, int st, int ed) {//----检查s从下标st开始到下标ed结束的字符串是否是回文串
    while (st < ed) {
        if (s[st++] != s[ed--]) return false;
    }
    return true;
}
vector<string>ans;
void dfs(string s, int st, int ed) {
    if (st > ed) {
        for (auto v : ans) cout << v << ' ';
        cout << endl;
    }
    for (int i = st; i <= ed; ++i) {
        if (check(s, st, i)) {
            //----将答案纳入数组
            ans.push_back(s.substr(st, i - st + 1));
            dfs(s, i + 1, ed);//----继续向下搜
            ans.pop_back();//----回溯
        }
    }
}
int main() {
    string s; cin >> s;
    dfs(s, 0, s.size() - 1);
    return 0;
}
```

查找一个队列中位数 分治策略

- https://blog.csdn.net/qq_40401156/article/details/105204168
- 采用快排的步骤来进行
- 按照快排的原理，选取一个 $base$,然后经过一系列处理后，$base$ 左边的数$\leq$ 右边的数
- 此时判断一下 $base$ 的当前位置，若是所求的位置则返回答案
- 若所求的位置 > $base$ 当前位置，则选取右边那块继续递归
- 同理若所求的位置 < $base$ 当前位置，则选取左边那块继续递归

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int num[N];
int solve(int k, int l, int r) {
    int idx = rand() % (r - l + 1) + l;
    int base = num[idx];
    int i = l, j = r;
    while (i < j) {
        while (j > i && num[j] >= base) --j;
        while (i < j && num[i] <= base) ++i;
        swap(num[i], num[j]);
    }
    swap(num[idx], num[i]);
```

```
    if (i == k) return base;
    else if (i < k) return solve(k, i + 1, r);
    else return solve(k, l, i - 1);
}
int main() {
    srand(time(NULL));
    int n; cin >> n;
    for (int i = 0; i < n; ++i) cin >> num[i];
    if (n % 2 == 1) cout << solve(n / 2, 0, n - 1) << '\n';
    else cout << 1.0 * (solve(n / 2 - 1, 0, n - 1) + solve(n / 2, 0, n - 1))
/ 2 << '\n';
}
```

01背包问题 动态规划

- 确认子问题和状态
  01背包问题需要求解的就是，为了体积V的背包中物体总价值最大化，N件物品中第i件应该放入背包中吗？（其中每个物品最多只能放一件）
  　　为此，我们定义一个二维数组，其中每个元素代表一个状态，即前i个物体中若干个放入体积为V背包中最大价值。数组为：$f[N][V]$，其中$f[i][j]$表示前i件中若干个物品放入体积为j的背包中的最大价值。
- 初始状态
  初始状态为$f[0][0-V]$和$f[0-N][0]$都为0，前者表示前0个物品（也就是空物品）无论装入多大的包中总价值都为0，后者表示体积为0的背包啥价值的物品都装不进去。
- 转移函数

```
if (背包体积j小于物品i的体积)
    f[i][j] = f[i-1][j] //背包装不下第i个物体，目前只能靠前i-1个物体装包
else f[i][j] = max(f[i-1][j], f[i-1][j-Vi] + W[i])
```

- 最后一句的意思就是根据“为了体积V的背包中物体总价值最大化，N件物品中第i件应该放入背包中吗？”转化而来的。$V[i]$表示第i件物体的体积，$W[i]$表示第i件物品的价值。这样$f[i-1][j]$代表的就是不将这件物品放入背包，而$f[i-1][j-V[i]]+W[i]$则是代表将第i件放入背包之后的总价值，比较两者的价值，得出最大的价值存入现在的背包之中。
  我们还可以采用滚动数组将二维数组优化到一维数组，降低了空间复杂度。

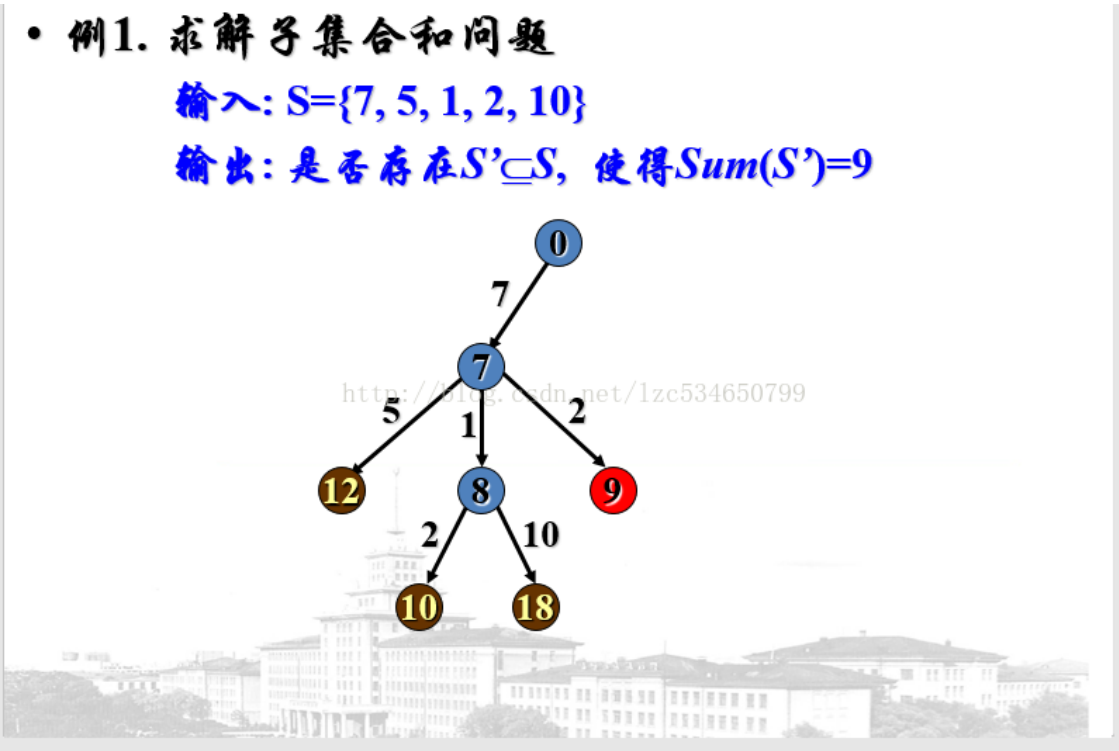
```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1e3 + 10;

int v[maxn], w[maxn], dp[maxn];
int n, m;

int main() {
    while(~scanf("%d %d", &n, &m)){
        memset(dp,0,sizeof(dp));
        for(int i = 0; i < n; i++)
            scanf("%d %d", &v[i], &w[i]);
        for(int i = 0; i < n; i++){
            for(int j = m; j >= 0; j--){
              if(j >= w[i]){
```

```
                dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
            }
        }
    }
    printf("%d\n", dp[m]);
  }
  return 0;
}
```

子集合问题 回溯

- 定义一个数组 $a[N]$，以及查找子集和为 k
- 

- ```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e2 + 10;
int a[N], sum[N];
int ans[N], cnt;
int flag=0;
void dfs(int pos,int n,int k) {
    if (pos > n || sum[n] - sum[pos - 1] < k) return;//----若区间[pos,n]的所有数字和都<k，则后面的一段无需在遍历
    if (k == 0) {
        flag = 1;
        for (int i = 1; i <= cnt; ++i) printf("%d%s", ans[i], i == cnt ? "\n" : " ");
        return;
    }
    for (int i = pos; i <= n; ++i) {
        if (a[i] > k) continue;
        ans[++cnt] = a[i];
        dfs(i + 1, n, k - a[i]);
        --cnt;//----回溯
```
```

```
    }
}
int main() {
    int n, k; cin >> n >> k;
    for (int i = 1; i <= n; ++i) cin >> a[i];
    for (int i = 1; i <= n; ++i) sum[i] = sum[i - 1] + a[i];//----计算前缀和，用于剪枝
    cnt = 0;
    dfs(1, n, k);
    if(!flag) printf("No Solution\n");
}
```

求最长上升子序列长度 动态规划

- 假定 $f[i]$ 表示下标为 i 权值，$dp[i]$ 表示 下标为 i 的当前最大上升序列长度

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N = 1e2 + 10;
int f[N], dp[N];
int main() {
    int n; cin >> n;
    for (int i = 1; i <= n; ++i)
        cin >> f[i];
    for (int i = 1; i <= n; ++i) dp[i] = 1;//----初始化所有值的当前最大上升子序列只有本身，即 1
    for (int i = 2; i <= n; ++i) {
        for (int j = 1; j < i; ++j) {
            if (f[j] <= f[i])
                dp[i] = max(dp[j] + 1, dp[i]);
        }
    }
    int ans = 0;
    for (int i = 1; i <= n; ++i)
        ans = max(ans, dp[i]);
    cout << ans << '\n';
}
```