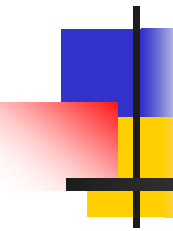
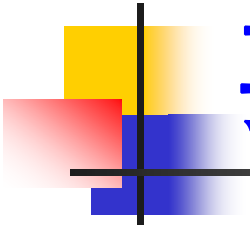


算法分析与设计

Analysis and Design of Algorithm

Lesson 07



要点回顾

- 分治策略
- 递推方程求解方法
  - 迭代法
  - 换元迭代法
  - 公式法
  - 递归树
  - 主定理

Strassen矩阵乘法

A和B的乘积矩阵C中的元素C[i,j]定义为:

$$C[i][j]=\sum_{k=1}^{n}A[i][k]B[k][j]$$

分析：若依此定义来计算A和B的乘积矩阵C，则每计算C的一个元素C[i][j]，需要做n次乘法和n-1次加法。因此，算出矩阵C的n^2个元素所需的计算时间为$O(n^3)$

Strassen矩阵乘法

分治法：

使用与大整数相乘类似的技术，将矩阵A，B和C中每一矩阵都分块成4个大小相等的子矩阵。由此可将方程C=AB重写为：

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

由此可得：

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

复杂度分析

1）n=2，子矩阵阶为1，8次乘和4次加，直接求出；
2）子矩阵阶大于2，为求子矩阵积可继续分块，直到子矩阵阶降为2。

此想法就产生了一个分治降阶递归算法。

两个n阶方阵的积➔8个$n/2$阶方阵积和4个$n/2$阶方阵加。

可在$O(n^2)$时间内完成

计算时间耗费 $$T(n)=\begin{cases} O(1) & n=2 \\ 8T(n/2)+O(n^2) & n>2 \end{cases}$$

所以 $T(n)=O(n^3)$**，与原始定义计算相比并不有效。**

复杂度分析(cont.)

原因是此法没有减少矩阵的乘法次数！！！

下面从计算2个2阶方阵乘开始，研究减少乘法次数(小于8次)

$$
\begin{aligned}
M_1 &= A_{11}(B_{12} - B_{22}) \\
M_2 &= (A_{11} + A_{12})B_{22} \\
M_3 &= (A_{21} + A_{22})B_{11} \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
M_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}
$$

7次乘

$$
\begin{aligned}
C_{11} &= M_5 + M_4 - M_2 + M_6 \\
C_{12} &= M_1 + M_2 \\
C_{21} &= M_3 + M_4 \\
C_{22} &= M_5 + M_1 - M_3 - M_7
\end{aligned}
$$

复杂度分析(cont.)

所以 需要7次乘法
18次加减法 → $T(n)=\begin{cases} O(1) & n=2 \\ 7T(n/2)+O(n^2) & n>2 \end{cases}$

→ $T(n)=O(n^{\log 7})=O(n^{2.8076})$ ✓较大的改进

Strassen矩阵乘法

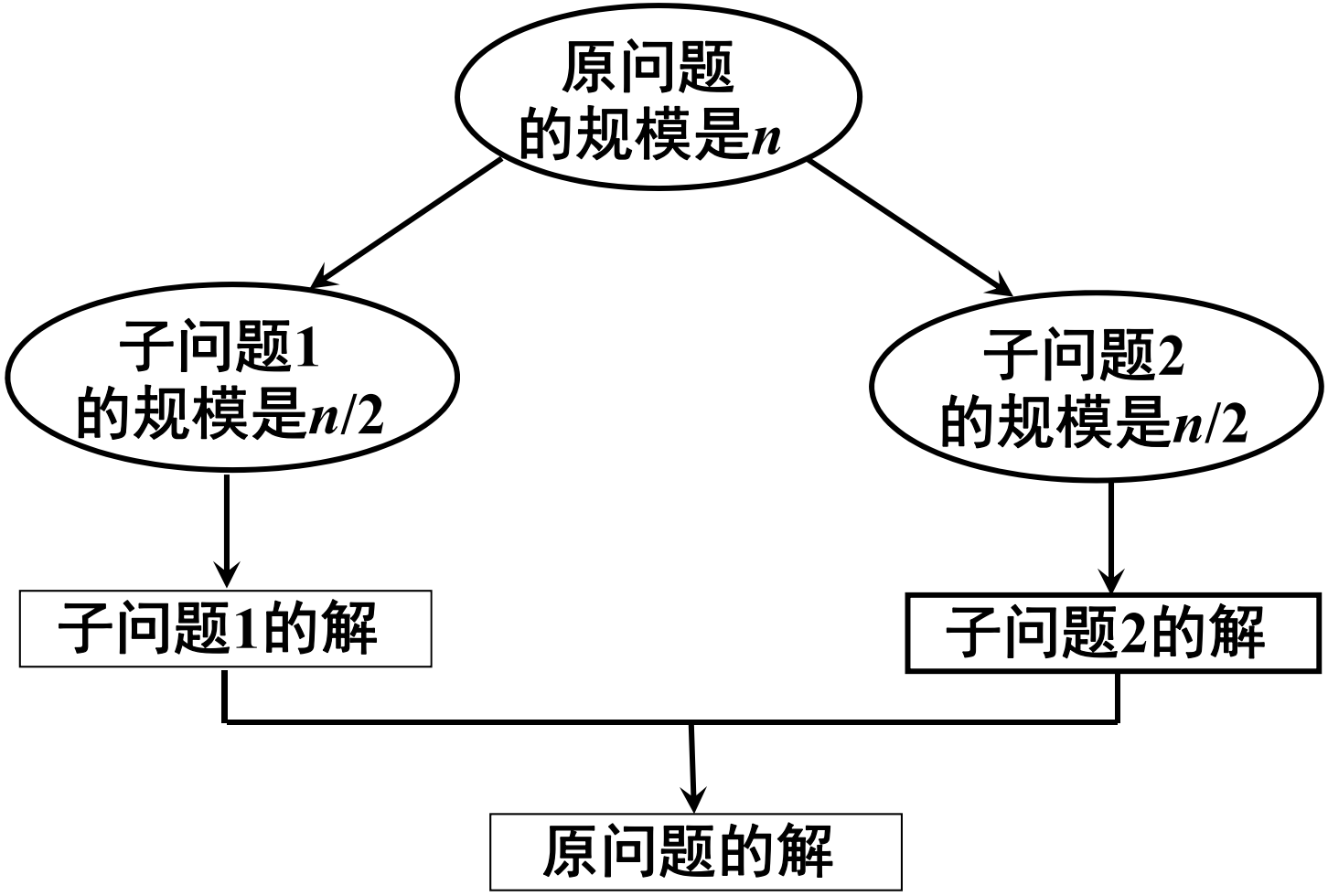
- 还有没有更快的方法?
- Hopcroft和Kerr已经证明(1971)，计算2个2×2矩阵的乘积，7次乘法是必要的。因此，要想进一步改进矩阵乘法的时间复杂性，就不能再基于计算2×2矩阵的7次乘法这样的方法了。或许应当研究3×3或5×5矩阵的更好算法。
- 在Strassen之后又有许多算法改进了矩阵乘法的计算时间复杂性。目前最好的计算时间上界是 $O(n^{2.3727})$

year	algorithm	order of growth
?	brute force	$O(n^3)$
1969	Strassen	$O(n^{2.808})$
1978	Pan	$O(n^{2.796})$
1979	Bini	$O(n^{2.780})$

是否能找到$O(n^2)$的算法?

1982	Romani	$O(n^{2.517})$
1982	Coppersmith-Winograd	$O(n^{2.496})$
1986	Strassen	$O(n^{2.479})$
1989	Coppersmith-Winograd	$O(n^{2.376})$
2010	Strother	$O(n^{2.3737})$
2011	Williams	$O(n^{2.3727})$

分治法的典型情况



排序问题中的分治法

二分归并排序/合并排序

二分归并排序的分治策略是：

- 划分：将待排序序列r_1, r_2, …, r_n划分为两个长度相等的子序列r_1, …, $r_{n/2}$和$r_{n/2+1}$, …, r_n；
- 求解子问题：分别对这两个子序列进行排序，得到两个有序子序列；
- 合并：将这两个有序子序列合并成一个有序序列。

二分归并排序/合并排序

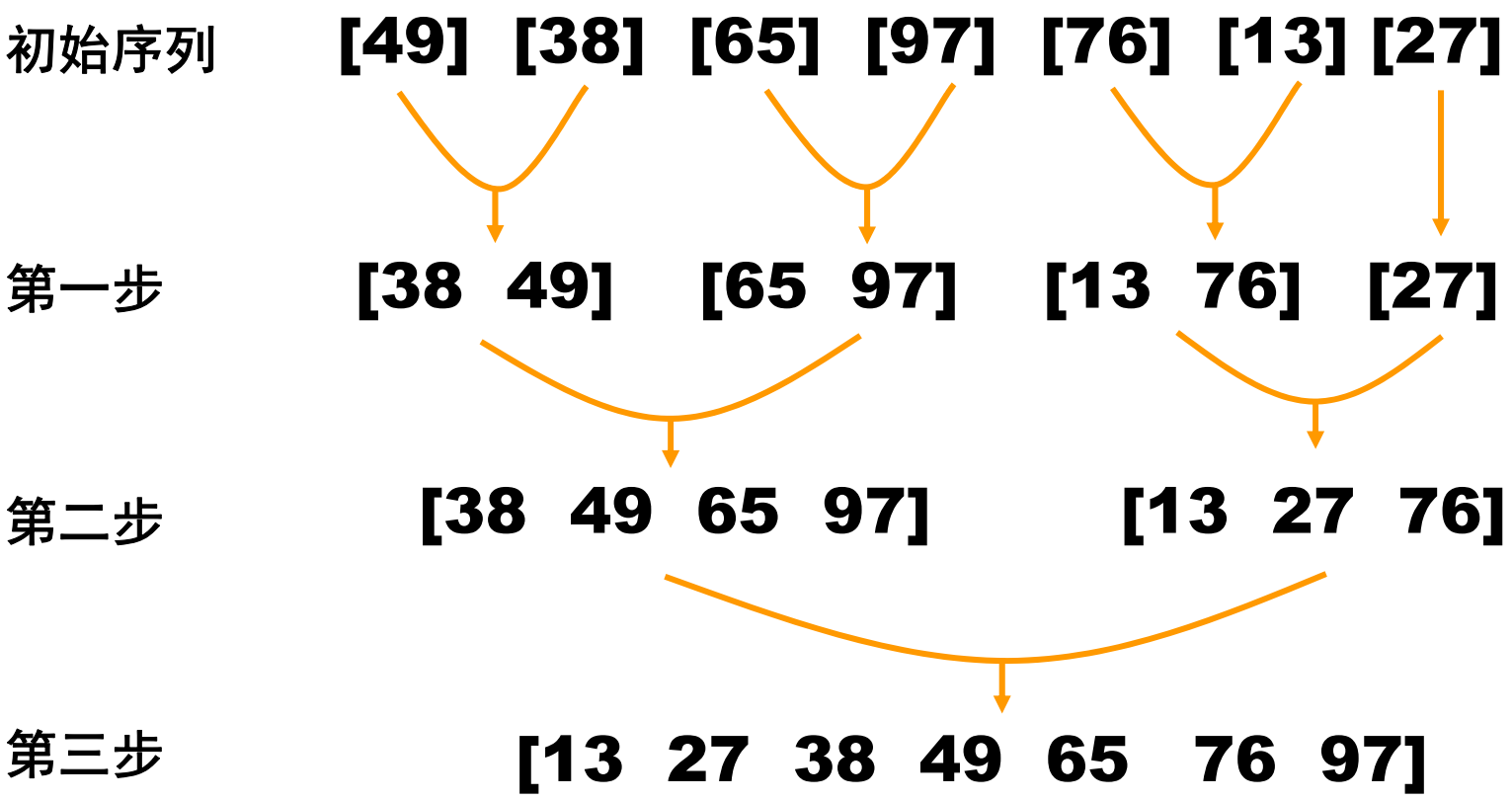
```
void MergeSort(Type a[], int left, int right){
     if (left<right) {//至少有2个元素
       int i=(left+right)/2; //取中点
       MergeSort(a, left, i);
       MergeSort(a, i+1, right);
       Merge(a, b, left, i, right); //合并到数组b
       Copy(a, b, left, right);     //复制回数组a
       }
}
```

复杂度分析 $T(n)=\begin{cases} O(1) & n \le 1 \\ 2T(n/2)+O(n) & n > 1 \end{cases}$

$T(n)=O(n\log n)$ 渐进意义下的最优算法

二分归并排序/合并排序

改进：算法mergeSort的递归过程可以消去。



二分归并排序/合并排序复杂度

- 最坏时间复杂度： $O(n\log n)$
- 平均时间复杂度：$O(n\log n)$
- 辅助空间：$O(n)$

可以看出，不论是最坏情况还是平均情况，二分归并排序的复杂度是最好的！且是个稳定排序。

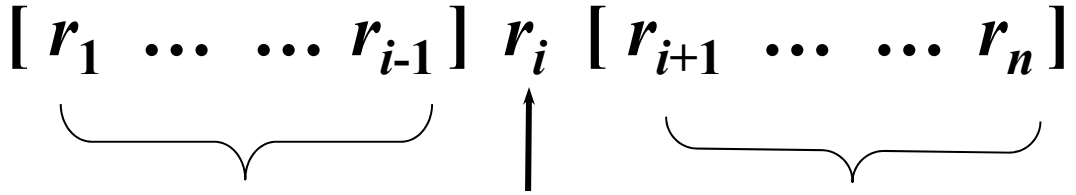
但它的一个重大缺点是，它不是一个就地操作的算法，需要O(n)个额外存储单元，当n很大的时候是一个很大的开销。

快速排序

快速排序的分治策略是：

- 划分：选定一个记录作为轴值，以轴值为基准将整个序列划分为两个子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$，前一个子序列中记录的值均小于或等于轴值，后一个子序列中记录的值均大于或等于轴值；
- 求解子问题：分别对划分后的每一个子序列递归处理；
- 合并：由于对子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$的排序是就地进行的，所以合并不需要执行任何操作。

快速排序



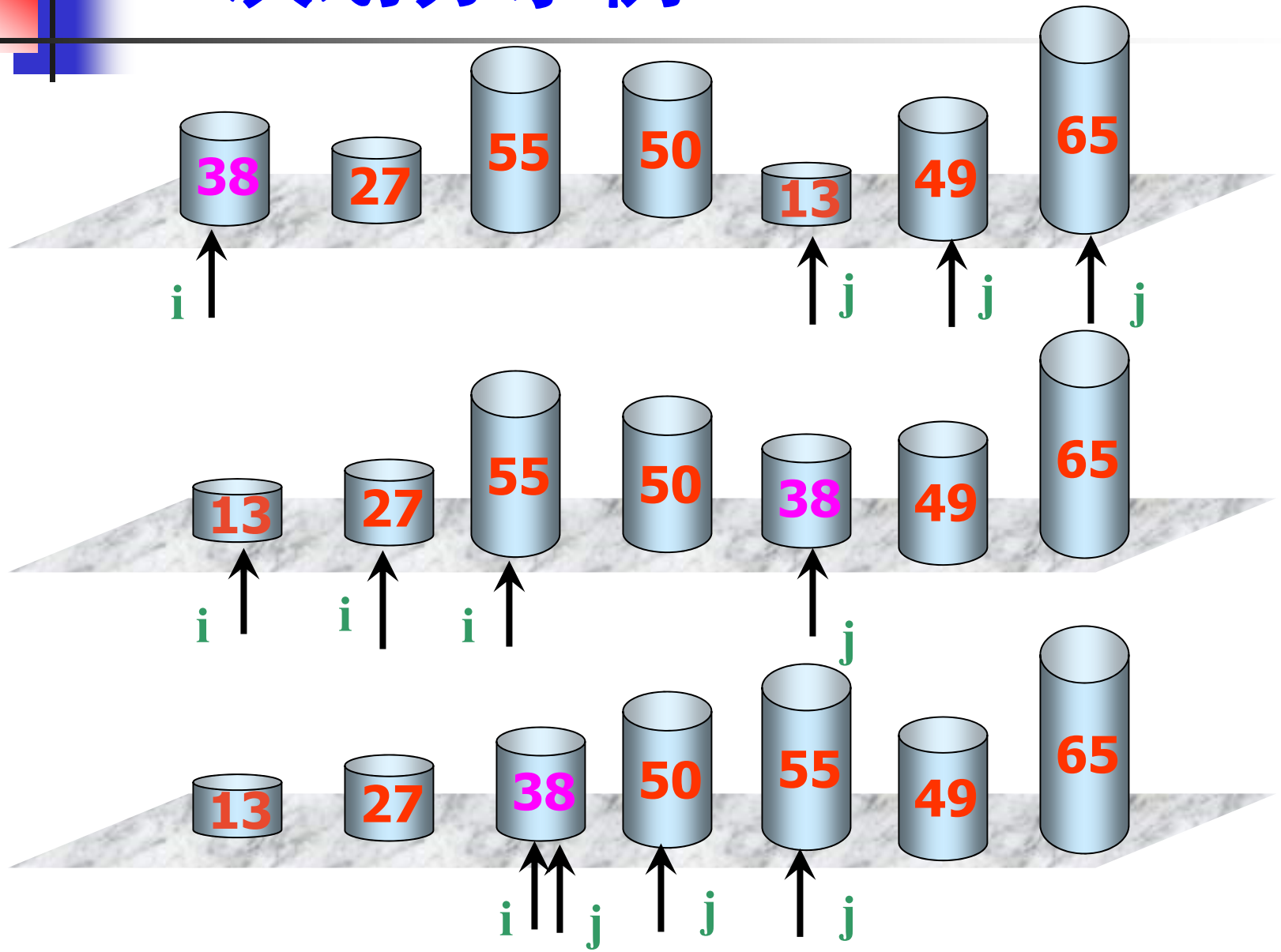
位于最终位置

- 归并排序按照元素在序列中的位置对序列进行划分，
- 快速排序按照元素的值对序列进行划分。

以第一个元素/记录作为轴值，对待排序序列进行划分的过程为：

1. 初始化：取第一个记录作为基准，设置两个参数i，j分别用来指示将要与基准记录进行比较的左侧记录位置和右侧记录位置，也就是本次划分的区间；
2. 右侧扫描过程：将基准记录与j指向的记录进行比较，如果j指向记录的关键码大，则j前移一个记录位置。重复右侧扫描过程，直到右侧的记录小（即反序）。若i＜j，则将基准记录与j指向的记录进行交换；
3. 左侧扫描过程：将基准记录与i指向的记录进行比较，如果i指向记录的关键码小，则i后移一个记录位置。重复左侧扫描过程，直到左侧的记录大（即反序）。若i＜j，则将基准记录与i指向的记录交换；
4. 重复2、3步，直到i与j指向同一位置，即基准记录最终的位置。

一次划分示例

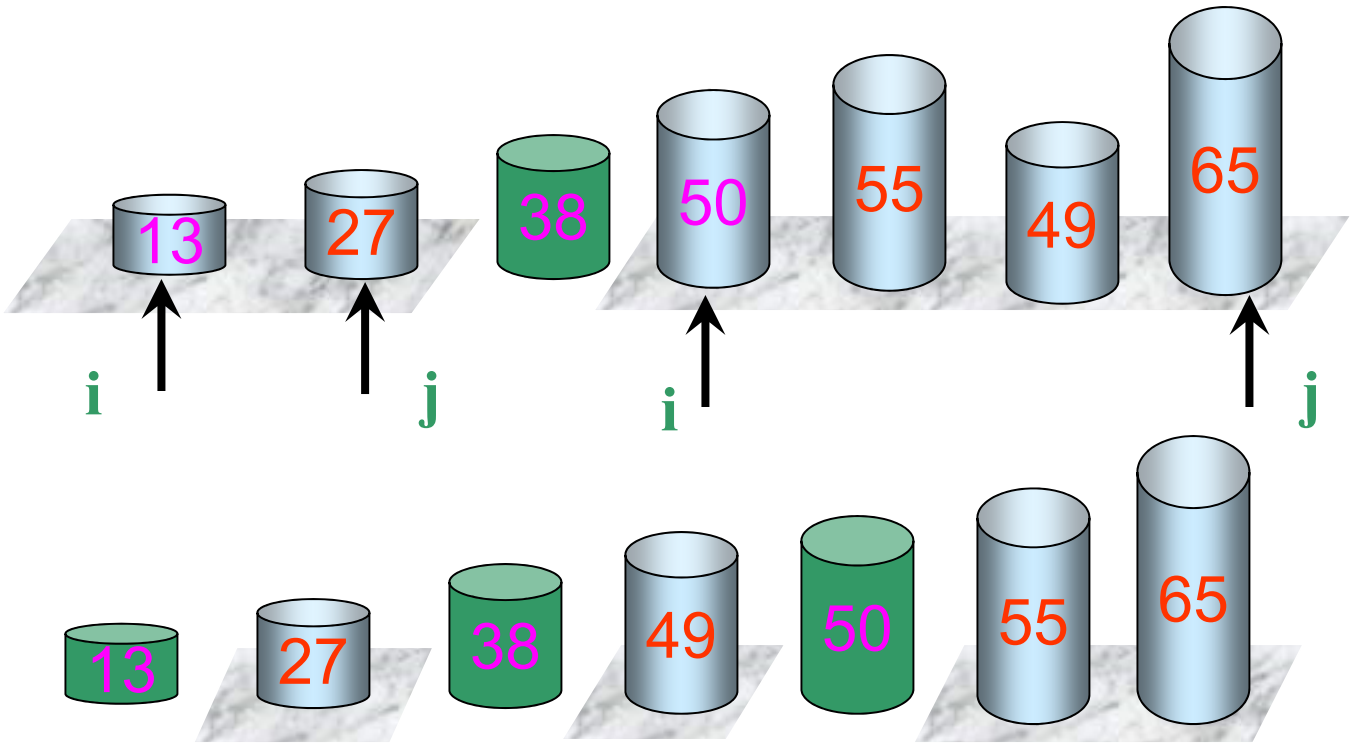

38
27
55
50
13
49
65
i
j
j
j
13
27
55
50
38
49
65
i
i
i
j
13
27
38
50
55
49
65
i
j
j
j

C++描述

一次划分算法伪代码

```
int Partition(int r[ ], int first, int end)
{
      i=first; j=end;              //初始化
      while (i<j){
           while (i<j && r[i]<= r[j]) j--;    //右侧扫描
           if (i<j) {
               r[i]←→r[j];                  //将较小记录交换到前面
               i++;
           }
          while (i<j && r[i]<= r[j]) i++;    //左侧扫描
          if (i<j) {
              r[j]←→r[i];                  //将较大记录交换到后面
              j--;
          }
      }
      return i;     // i为轴值记录的最终位置
}
```

以轴值为基准将待排序序列划分为两个子序列后，对每一个子序列分别递归进行排序。



C++描述

快速排序算法伪代码

```
void QuickSort(int r[ ], int first, int end){
    if (first<end) {
        pivot=Partition(r, first, end);
          //问题分解，pivot是轴值在序列中的位置
        QuickSort(r, first, pivot-1);
          //递归地对左侧子序列进行快速排序
        QuickSort(r, pivot+1, end);
         //递归地对右侧子序列进行快速排序
    }
}
```

在最好情况下，每次划分对一个记录定位后，该记录的左侧子序列与右侧子序列的长度相同。在具有n个记录的序列中，一次划分需要对整个待划分序列扫描一遍，则所需时间为$O(n)$。设$T(n)$是对n个记录的序列进行排序的时间，每次划分后，正好把待划分区间划分为长度相等的两个子序列，则有：

$$T(n) \leqslant 2\ T(n/2)+n$$

$$\leqslant 2(2T(n/4)+n/2)+n = 4T(n/4)+2n$$

$$\leqslant 4(2T(n/8)+n/4)+2n = 8T(n/8)+3n$$

$$\cdots\cdots\cdots$$

$$\leqslant nT(1)+n\log_2 n = O(n\log n)$$

因此，时间复杂度为$O(n\log n)$。

在最坏情况下，待排序记录序列正序或逆序，每次划分只得到一个比上一次划分少一个记录的子序列（另一个子序列为空）。此时，必须经过n-1次递归调用才能把所有记录定位，而且第i趟划分需要经过n-i次关键码的比较才能找到第i个记录的基准位置，因此，总的比较次数为：

$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)=O(n^2)$$

因此，时间复杂度为$O(n^2)$。

在平均情况下，设基准记录的关键码第k小($1 \leqslant k \leqslant n$)，则有：

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}(T(n-k)+T(k-1))+n=\frac{2}{n}\sum_{k=1}^{n}T(k)+n$$

这是快速排序的平均时间性能，可以用归纳法证明，其数量级也为$O(n\log n)$。

快速排序复杂度分析小结

- 快速排序算法的性能取决于划分的对称性。
- 快速排序时间与划分是否对称有关，最坏情况一边n-1个，一边1个。
  - 如不对称，则：$T(n)=\begin{cases}O(1) & n\le 1\\ T(n-1)+O(n) & n>1\end{cases}$

    解得 $T(n)=O(n^2)$
  - 最好情况，对称，则：$T(n)=\begin{cases}O(1) & n\le 1\\ 2T(n/2)+O(n) & n>1\end{cases}$

    解得 $T(n)=O(n\log n)$

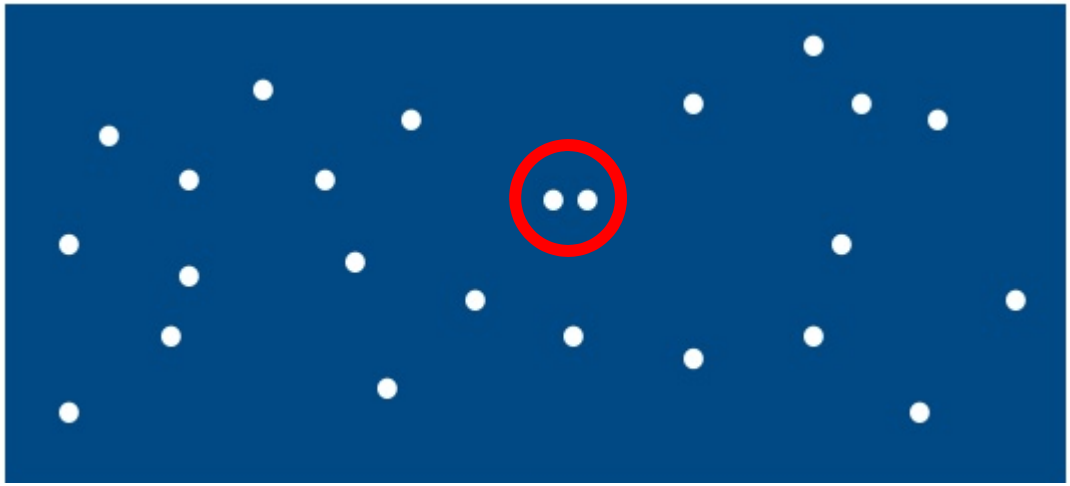
结论： 最坏时间复杂度：$O(n^2)$
平均时间复杂度：$O(n\log n)$

几何问题中的分治法
（自学）

最接近点对问题

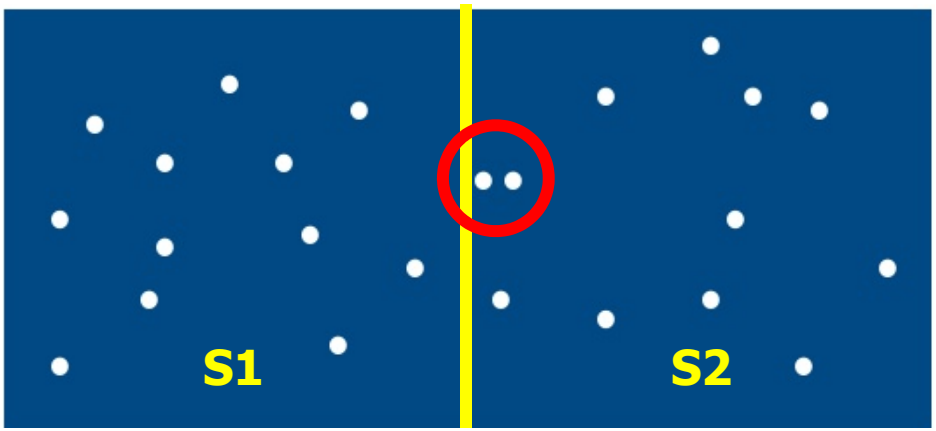
设$p_1=(x_1, y_1)$, $p_2=(x_2, y_2)$, …, $p_n=(x_n, y_n)$是平面上n个点构成的集合S，最近对问题就是找出集合S中距离最近的点对。

严格地讲，最接近点对可能多于一对，简单起见，只找出其中的一对作为问题的解。



最接近点对问题

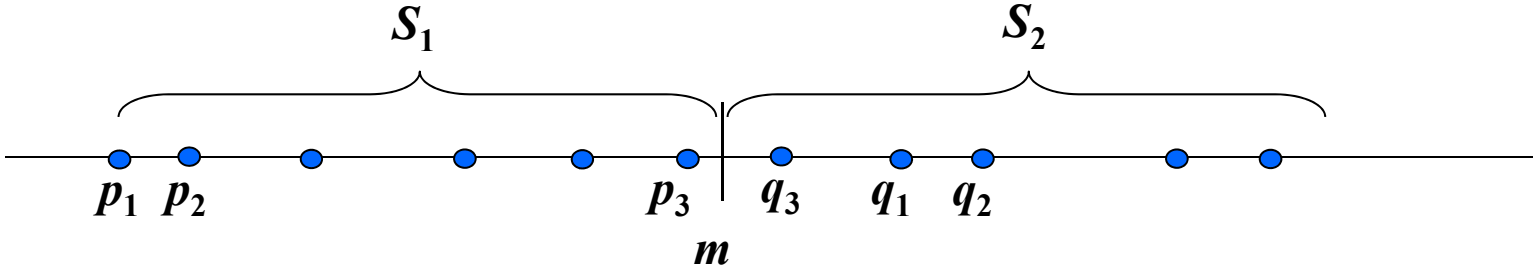
用分治法解决最近对问题，很自然的想法就是将集合S分成两个子集 $S1$和 $S2$，每个子集中有$n/2$个点。然后在每个子集中递归地求其最接近的点对，在求出每个子集的最接近点对后，在合并步中，如果集合 S 中最接近的两个点都在子集 $S1$或 $S2$中，则问题很容易解决，如果这两个点分别在 $S1$和 $S2$中，问题就比较复杂了。



为了使问题易于理解，先考虑一维的情形。

此时，S中的点退化为x轴上的n个点$x_1, x_2, \ldots, x_n$。用x轴上的某个点m将S划分为两个集合S_1和S_2，并且S_1和S_2含有点的个数相同。递归地在S_1和S_2上求出最接近点对 (p_1, p_2) 和 (q_1, q_2)，如果集合S中的最接近点对都在子集S_1或S_2中，则 $d=\min\{(p_1, p_2), (q_1, q_2)\}$即为所求。

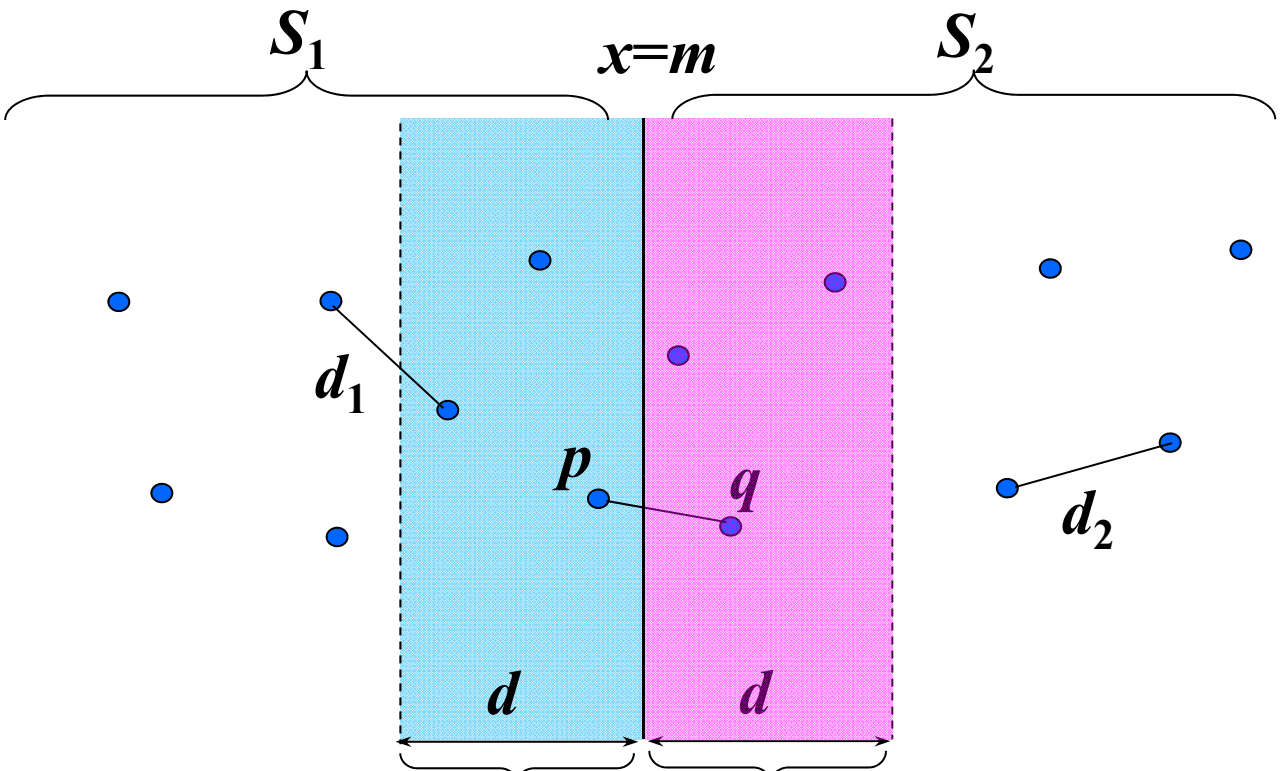
如果集合S中的最接近点对分别在S_1和S_2中，则一定是(p_3, q_3)，其中，p_3是子集S_1中的最大值，q_3是子集S_2中的最小值。



下面考虑二维的情形，此时S中的点为平面上的点。

- 为了将平面上的点集S 分割为点的个数大致相同的两个子集S_1和S_2，选取垂直线$x=m$来作为分割线，其中，m为S中各点x坐标的中位数。
- 由此将S分割为$S_1=\{p\in S \mid x_p\leqslant m\}$和$S_2=\{q\in S \mid x_q>m\}$。
- 递归地在S_1和S_2上求解最近对问题，分别得到S_1中的最近距离d_1和S_2中的最近距离d_2。
- 令$d=\min(d_1, d_2)$，若S的最近点对(p, q)之间距离小于d，则p和q必分属于S_1和S_2，不妨设$p\in S_1$，$q\in S_2$，则p和q距直线$x=m$的距离均小于d，所以，可以将求解限制在以$x=m$为中心、宽度为$2d$的垂直带$P1$和$P2$中，垂直带之外的任何点对之间的距离都一定大于d。

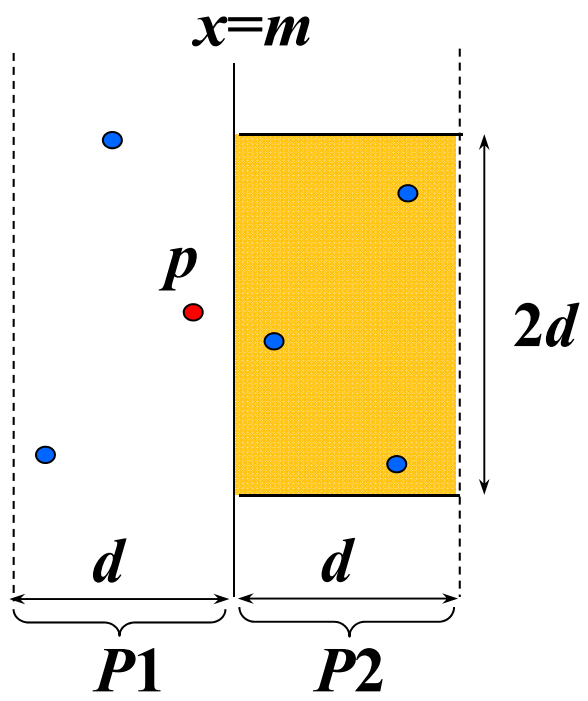
最近对问题的分治思想



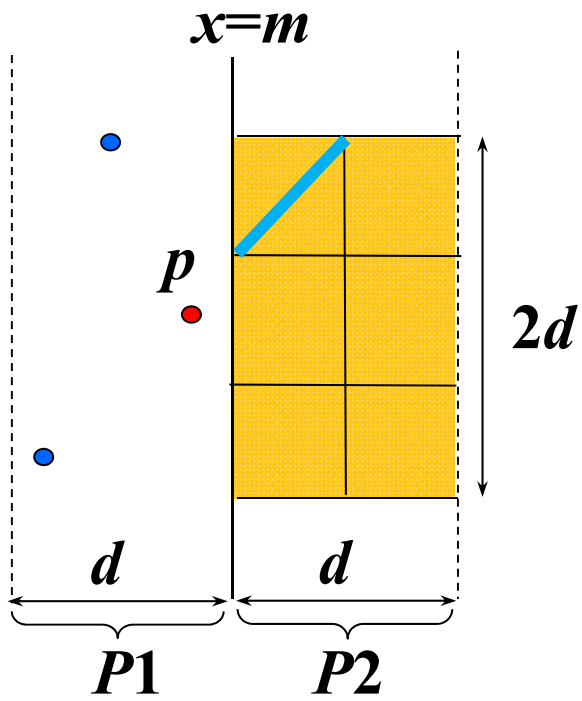
$S_1=\{p\in S \mid x_p\leqslant m\}$

$S_2=\{q\in S \mid x_q>m\}$

对于点$p \in P1$，需要考察$P2$中的各个点和点p之间的距离是否小于d，显然，$P2$中这样点的y轴坐标一定位于区间$[y-d, y+d]$之间。而且，这样的点不会超过？个。



(a) 包含点q的$d \times 2d$的矩形区域



(b) 最坏情况下需要检查的6个点

最接近点对问题复杂度分析

应用分治法求解含有n个点的最近对问题，其时间复杂性可由下面的递推式表示：

$$T(n) = 2T(n/2) + f(n)$$

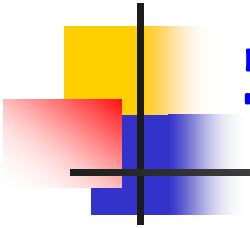
合并子问题的解的时间$f(n)=O(1)$，根据主定理，可得$T(n)=O(n\log n)$。

第二章要点回顾

- 递归算法
  - 概念（阶乘、Fibonacci数列、双递归）
  - 例子（整数划分问题、Hanoi塔问题）
  - Hanoi塔算法、运行轨迹、分析时间复杂度
  - 递推方程（迭代法求解）
  - 递归的优缺点
- 分治策略
  - 基本思想、适用条件、基本步骤
  - 分治效率分析：给出了五种计算方法
  - 范例学习：二分搜索、大整数乘法、Strassen矩阵乘法、合并排序、快速排序、最近点对问题（自学）

第三章 动态规划法

Dynamic Programming



学习要点

- 理解动态规划算法的概念
- 掌握动态规划算法的基本要素
  - 最优子结构性质
  - 重叠子问题性质
- 掌握设计动态规划算法的步骤
  - 找出最优解的性质，并刻划其结构特征
  - 递归地定义最优值
  - 以自底向上的方式计算出最优值
  - 根据计算最优值时得到的信息，构造最优解
- 动态规划法应用实例

动态规划算法概述

概述

1. 最优化问题
2. 最优性原理
3. 动态规划法的设计思想

1.最优化问题

- 有n个输入（解空间），问题的解由这n个输入的一个子集组成，这个子集必须满足某些事先给定的条件，这些条件称为约束条件，满足约束条件的解称为问题的可行解。
- 满足约束条件的可行解可能不只一个，为了衡量这些可行解的优劣，事先给出一定的标准，这些标准通常以函数的形式给出，称为目标函数，使目标函数取得极值（极大或极小）的可行解称为最优解。
- 这类问题就称为最优化问题。

例：找零钱问题

问题： 自动售货POS机要找给顾客数量最少的现金。

分析： 假定POS机中有n张面值为$p_i(1 \leqslant i \leqslant n)$的货币，用集合$P=\{p_1, p_2, \ldots, p_n\}$表示，如果POS机需找的现金为$A$，那么，必须从$P$中选取一个最小子集$S$，使得：

$$p_i \in S \quad , \quad \sum_{i=1}^{m} p_i = A \quad (m = |S|) \qquad \text{式(1)}$$

如果用向量$X=(x_1, x_2, \ldots, x_n)$，表示$S$中所选取的货币，则：

$$x_i = \begin{cases} 1 & p_i \in S \\ 0 & p_i \notin S \end{cases} \qquad \text{式(2)}$$

例：找零钱问题

那么，POS机找的现金必须满足 $\sum_{i=1}^{n} x_i p_i = A$ 式(3)

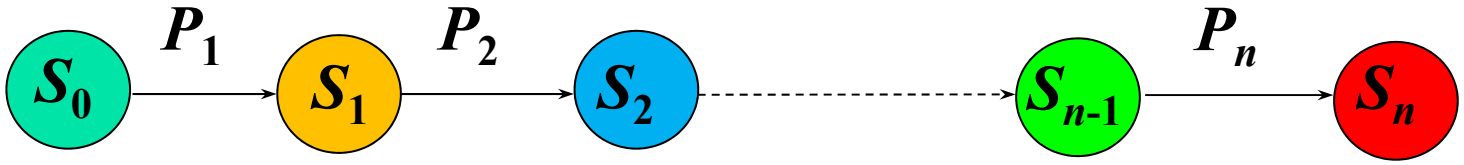
并且 $d = \min \sum_{i=1}^{n} x_i$ 式(4)

在找零钱问题中，集合*P*是该问题的输入，那么：

- 满足式(1)的解称为可行解；
- 式(2)是解的表现形式(因为向量*X*中有*n*个元素，每个元素的取值为0或1，所以，可以有2^n个不同的向量，所有这些向量的全体构成该问题的解空间)；
- 式(3)是该问题的约束条件；
- 式(4)是该问题的目标函数；
- 使式(4)取得极小值的解称为该问题的最优解。

2.最优性原理

对于一个具有n个输入的最优化问题，其求解过程往往可以划分为若干个阶段，每一阶段的决策仅依赖于前一阶段的状态，由决策所采取的动作使状态发生转移，成为下一阶段决策的依据。从而，一个决策序列在不断变化的状态中产生。这个决策序列产生的过程称为多阶段决策过程。



多阶段决策过程

2.最优性原理

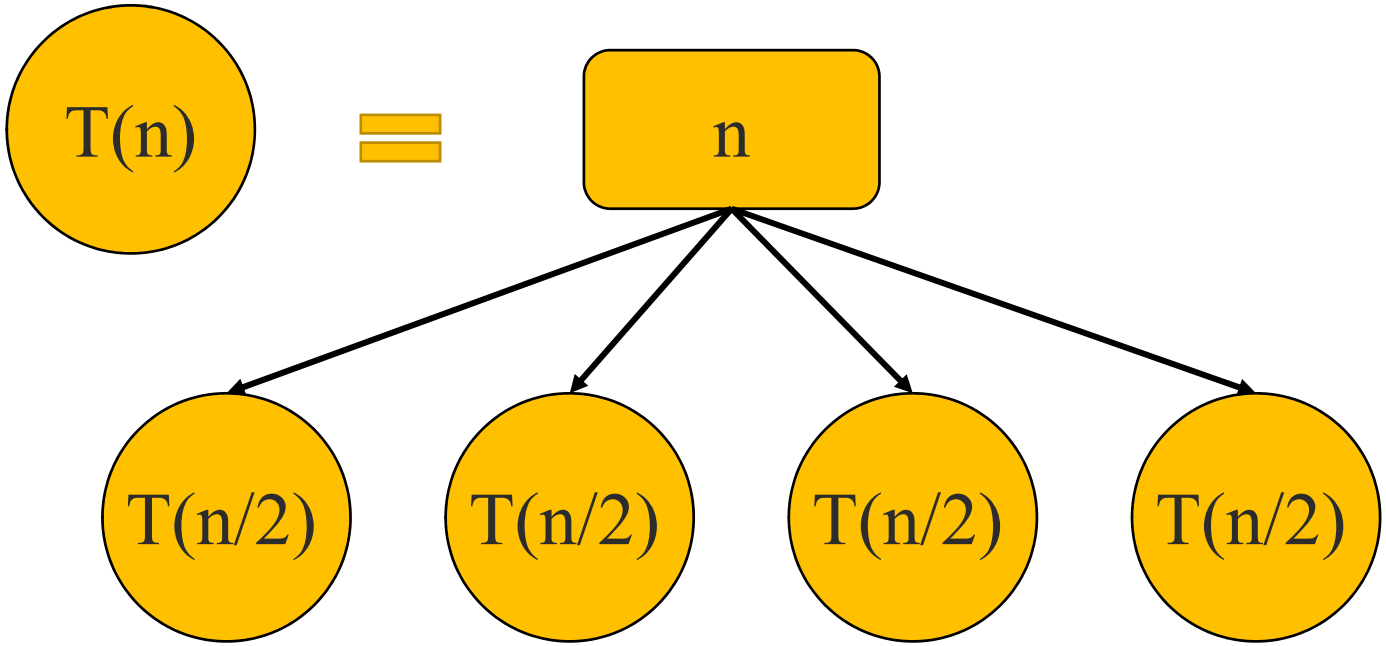
在每一阶段的决策中有一个赖以决策的策略或目标，这种策略或目标是由问题的性质和特点所确定，通常以函数的形式表示并具有递推关系，称为动态规划函数。

多阶段决策过程满足最优性原理或者优化原则(Optimal Principle)：无论决策过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的当前状态，构成一个最优决策序列。

如果一个问题满足最优性原理/优化原则通常称此问题具有最优子结构性质。

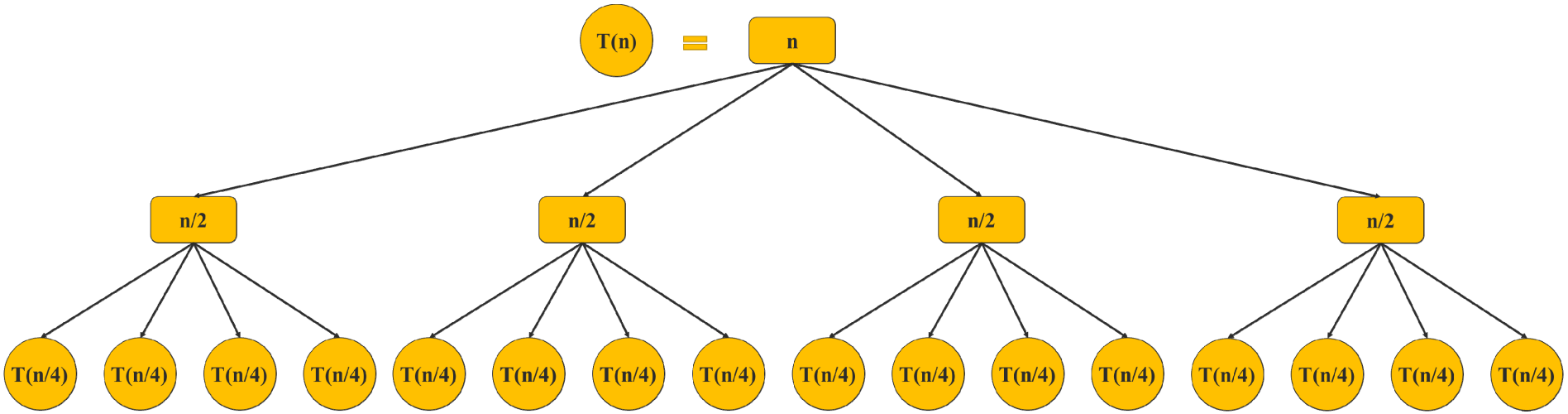
3.算法总体思想

- 动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题



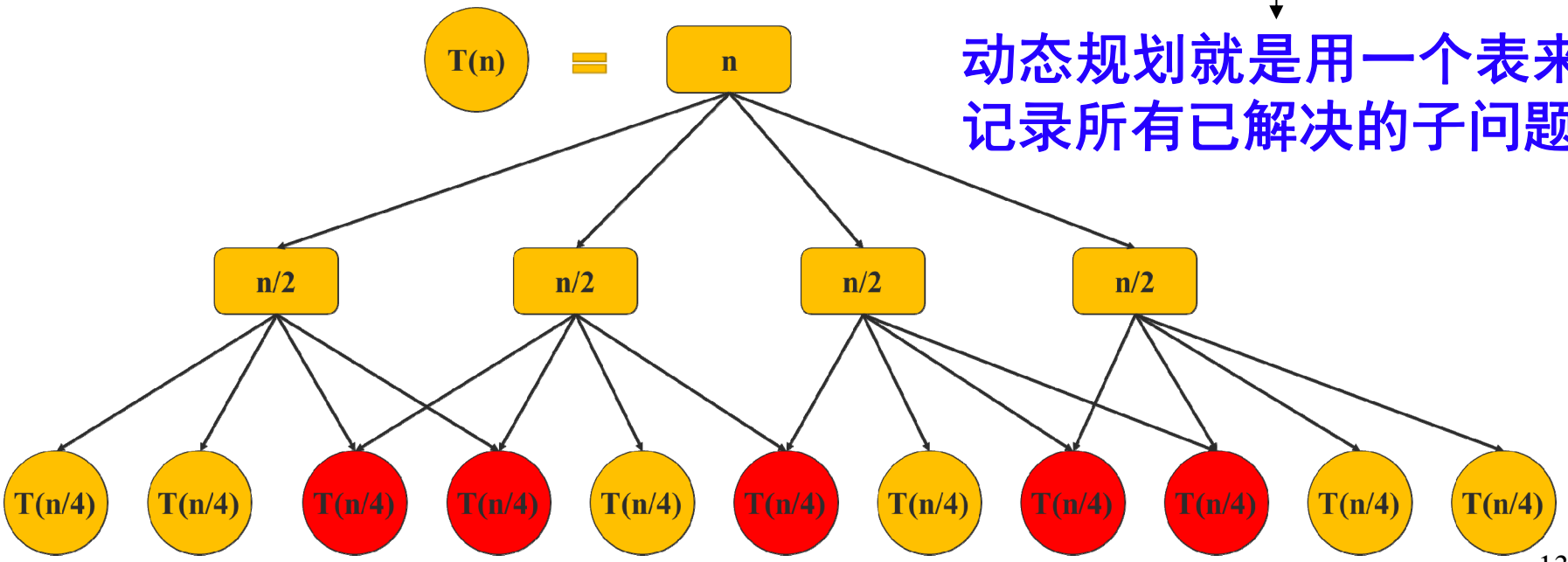
3.算法总体思想

- **与分治区别：**经分解得到的子问题往往不是互相独立的。不同子问题的数目常常只有多项式量级。如果用分治法求解，有些子问题被重复计算了许多次。



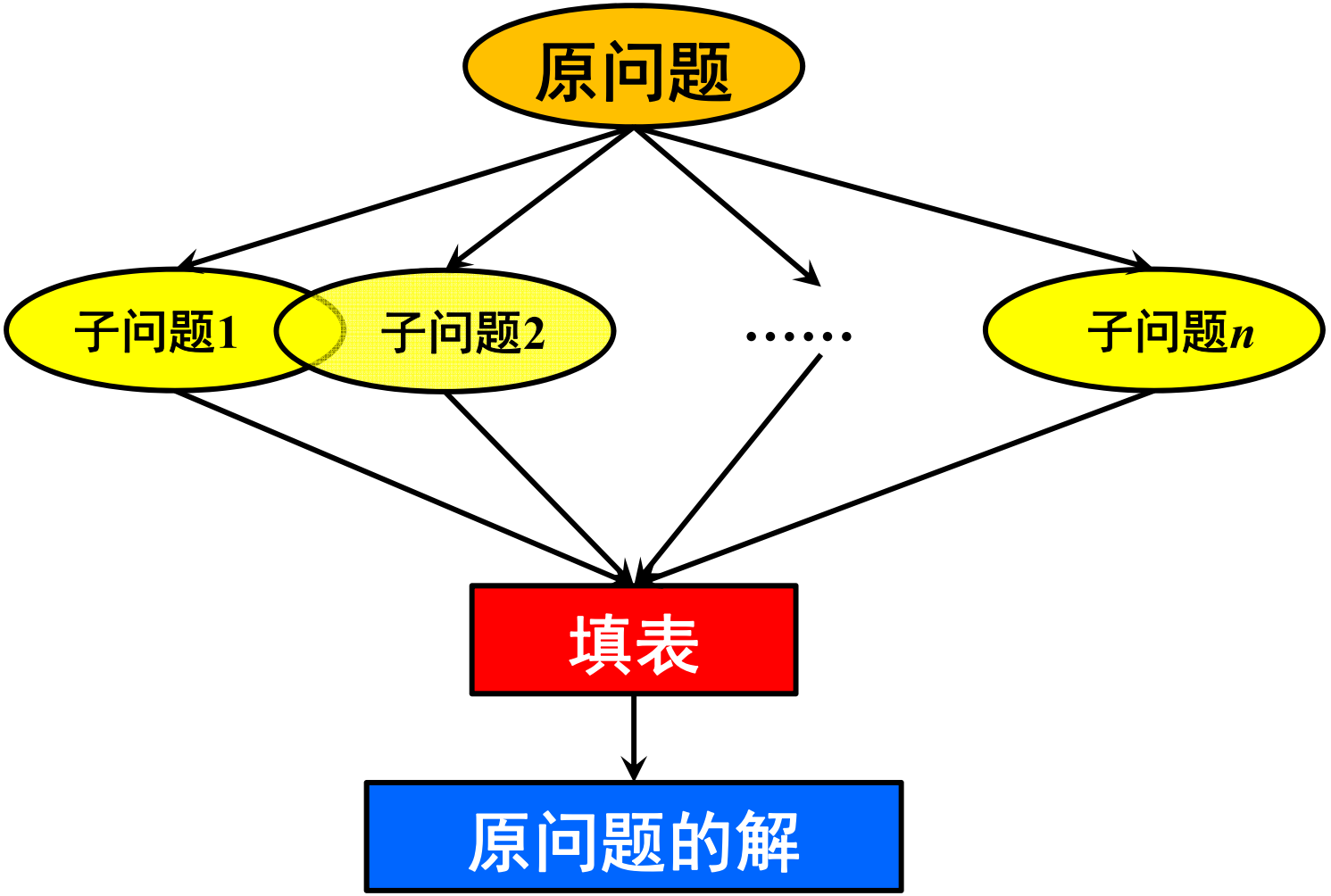
3.算法总体思想

- 用分治法求解，子问题的数目常有多项式量级，有些子问题被重复计算了多次，如果能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，就可以避免大量重复计算，从而得到多项式时间算法。

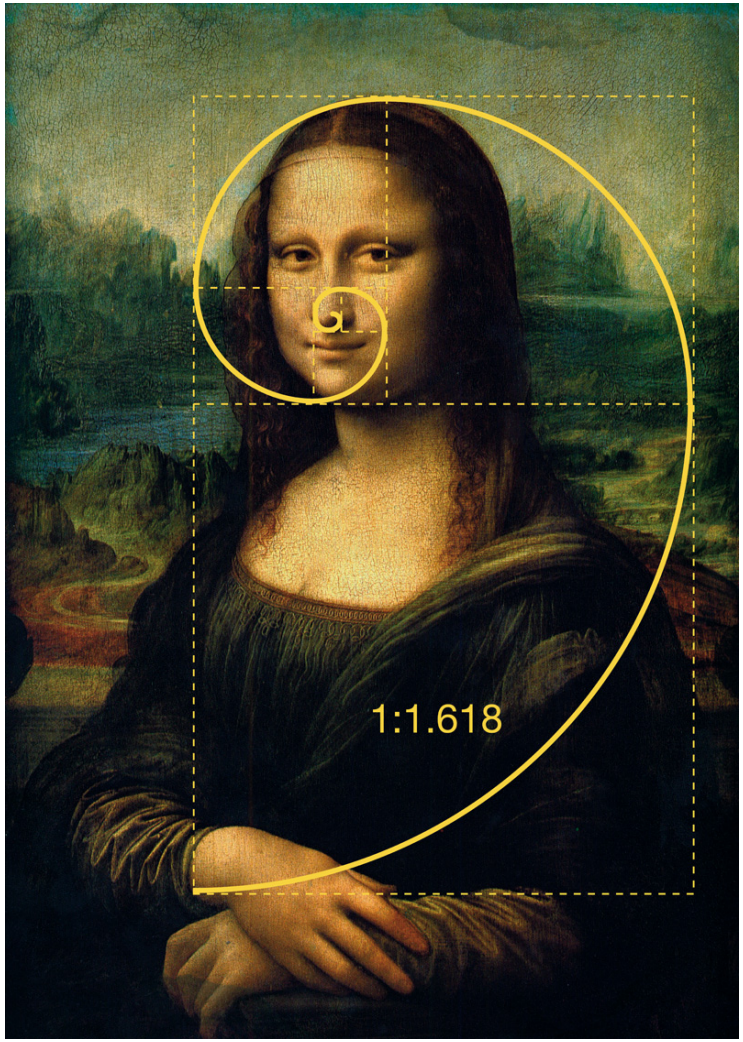


动态规划就是用一个表来记录所有已解决的子问题

动态规划法的求解过程

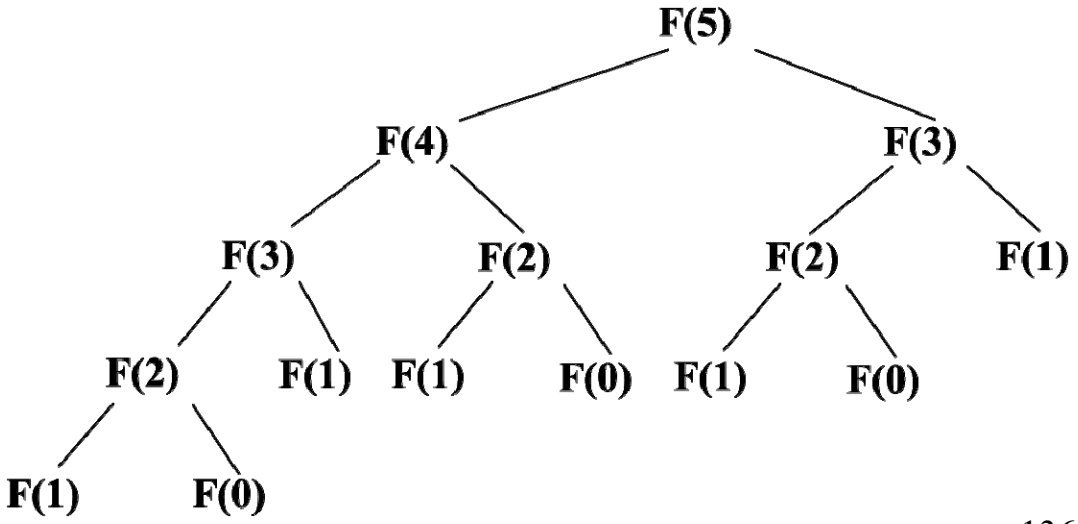


例：计算Fibonacci数列



$$F(n)=\begin{cases}1 & ,\ n=0\\ 1 & ,\ n=1\\ F(n-1)+F(n-2) & ,\ n>1\end{cases}$$

***n*=5时分治法计算斐波那契数的过程：**



例：计算Fibonacci数列

分析：注意到，计算$F(n)$是以计算它的两个重叠子问题$F(n-1)$和$F(n-2)$的形式来表达的，所以，可以设计一张表填入$n+1$个$F(n)$的值。

动态规划法求解斐波那契数$F(9)$的填表过程：

0	1	2	3	4	5	6	7	8	9
0	1	1	2	3	5	8	13	21	34

动态规划法注意事项

- 用动态规划法求解的问题具有特征：
  - 能够分解为相互重叠的若干子问题；
  - 满足最优性原理（也称最优子结构性质）：该问题的最优解中也包含着其子问题的最优解。
- 用反证法分析问题是否满足最优性原理：
  - 先假设由问题的最优解导出的子问题的解不是最优的；
  - 然后再证明在这个假设下可构造出比原问题最优解更好的解，从而导致矛盾。

动态规划法基本步骤

动态规划法设计算法一般分成三个阶段：

1. 分段：将原问题分解为若干个相互重叠的子问题；
2. 分析：分析问题是否满足最优性原理或者优化原则，找出动态规划函数的递推式；
3. 求解：利用递推式自底向上计算，实现动态规划过程。

记住：动态规划法利用问题的最优性原理，以自底向上的方式从子问题的最优解逐步构造出整个问题的最优解。

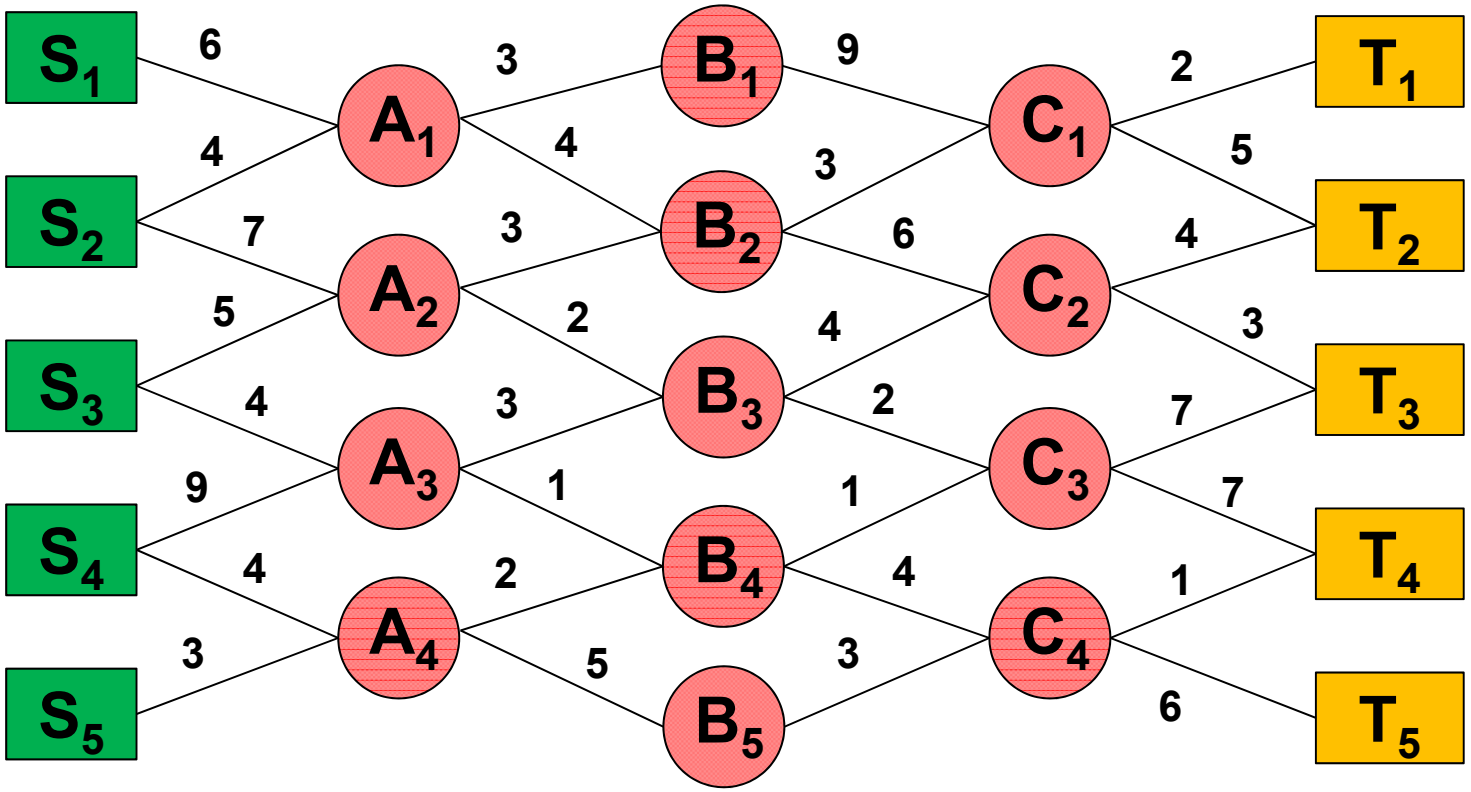
例：最短路径问题

- **问题：** 找任意起点到任意终点的一条最短路径

- **输入：**
  - 起点集合$\{S_1,S_2,\ldots,S_n\}$
  - 终点集合$\{T_1,T_2,\ldots,T_m\}$
  - 中间结点集合，边集，对于任意边e有长度
- **输出：** 一条从起点到终点的最短路

例：最短路径问题

- 一个实例

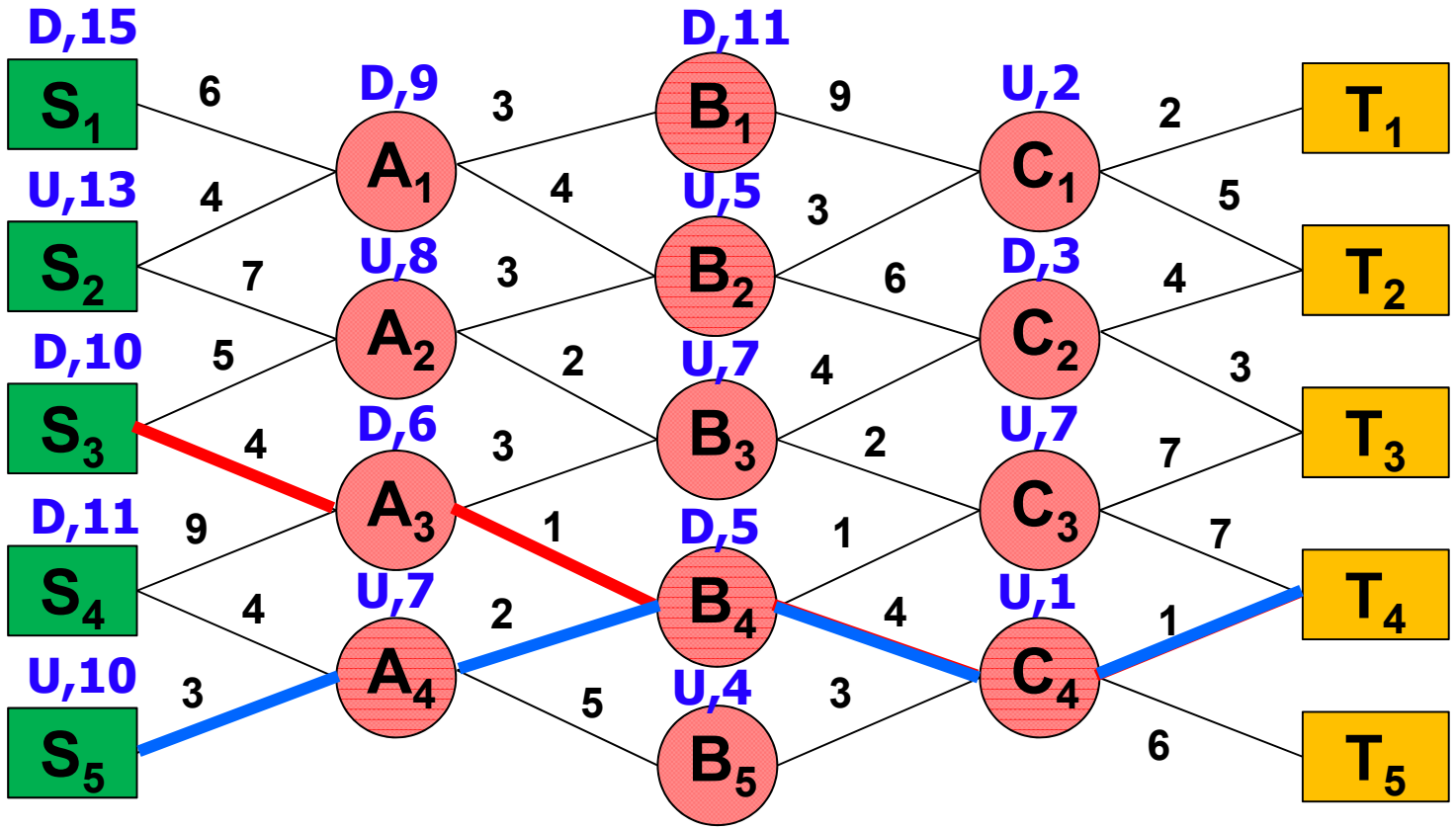


例：最短路径问题

- **蛮力算法/穷举法：**考察每一条从某个起点到某个终点的路径，计算长度，从中找出最短路径。
- 在上述实例中，如果网络的层数为k，那么路径条数将接近2^k
- **动态规划算法：**多阶段决策过程。每一步求解的问题是后面阶段求解问题的子问题。每步决策将依赖于以前步骤的决策结果。

例：最短路径问题

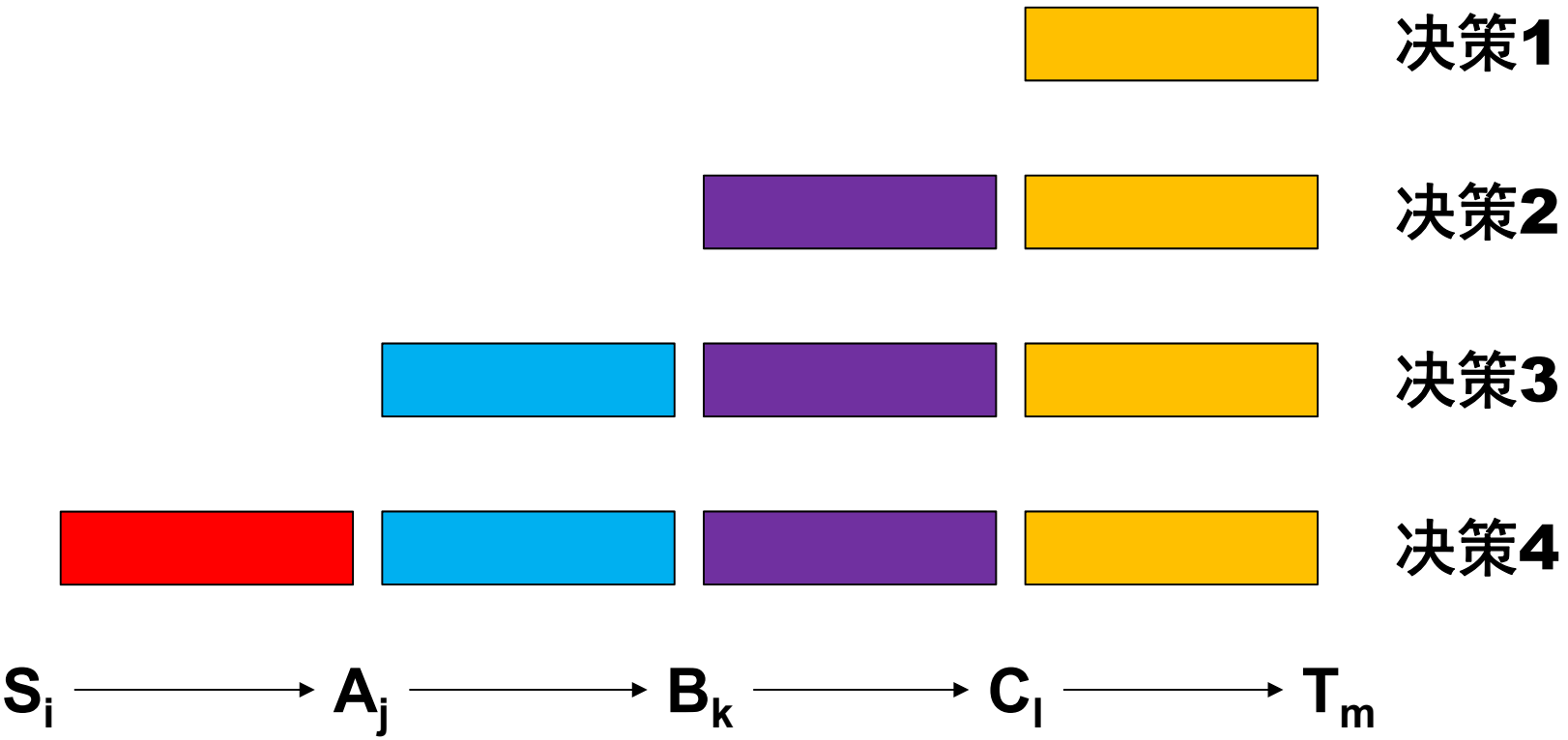
- 动态规划求解



例：最短路径问题

- 子问题界定

后边界不变，前边界前移



例：最短路径问题

- 最短路径的依赖关系

$$F(C_l)=\min_m\{C_lT_m\}$$ 决策1

$$F(B_k)=\min_l\{B_kC_l+F(C_l)\}$$ 决策2

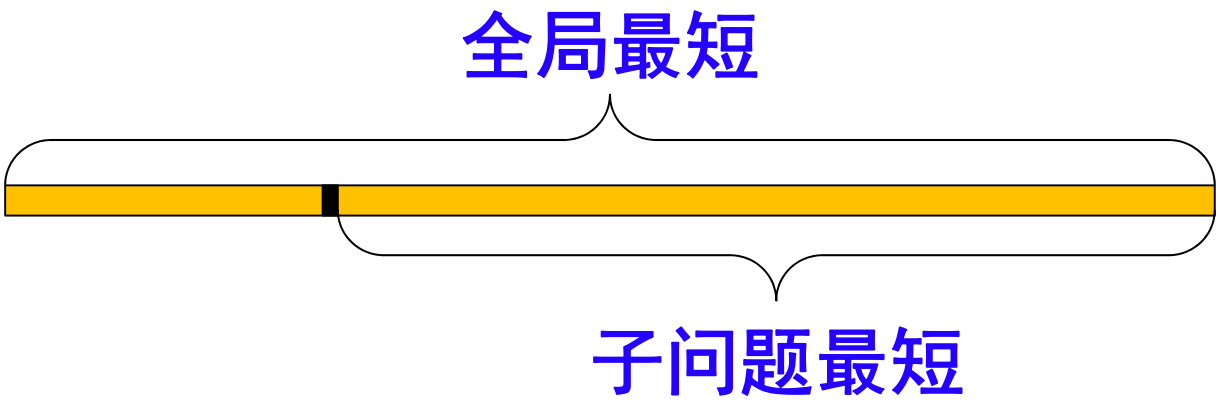
$$F(A_j)=\min_k\{A_jB_k+F(B_k)\}$$ 决策3

$$F(S_i)=\min_j\{S_iA_j+F(A_j)\}$$ 决策4

优化函数值之间存在依赖关系

优化原则：最优子结构性质

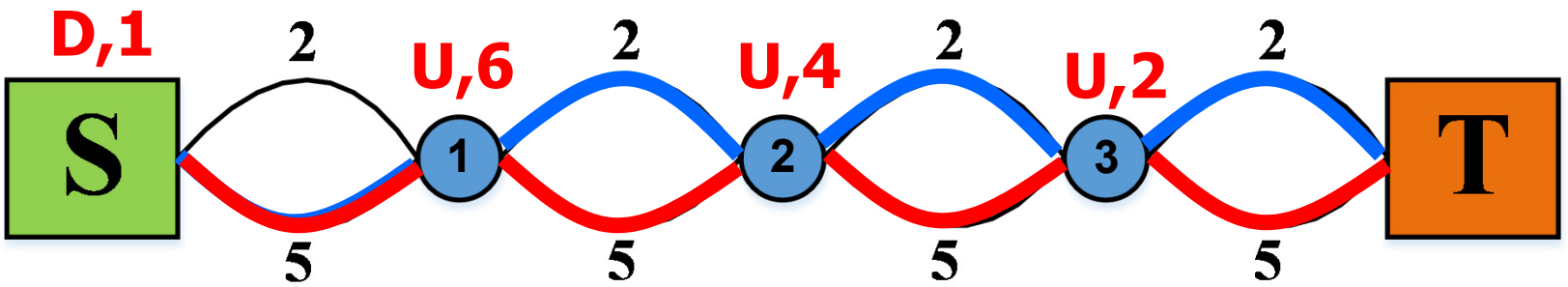
- **优化函数的特点：**任何最短路的子路径相对于子问题始、终点最短



- **优化原则：**一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列

一个反例

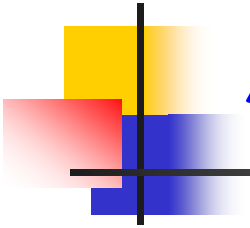
- 求总长模10的最小路径



➔动态规划算法的解：下、上、上、上

- 最优解：下、下、下、下

不满足优化原则，不能用动态规划！！！



小结

动态规划(Dynamic Programming)

- 求解过程是多阶段决策过程，每步处理一个子问题，可用于求解组合优化问题
- 适用条件：问题要满足优化原则或者最优子结构性质，即：一个最优决策序列的任何子序列本身一定是相对于子序列的初始和结束状态的最优决策序列

动态规划算法设计

动态规划设计要素

1. 问题建模，优化的目标函数是什么？约束条件是什么？
2. 如何划分子问题？（边界）
3. 问题的优化函数值与子问题的优化函数值存在着什么依赖关系？（递推方程）
4. 是否满足优化原则/最优子结构性质？
5. 最小子问题怎样界定？其优化函数值，即初值等于什么？