Algorithms Design and Analysis

Lecture 3

Beihang University

2017

Divide and Conquer (Continue)

Example 4. Finding Minimum (最小) and Maximum (最大)

Backgound (背景)

Find the lightest (最轻) and heaviest (最重) of n elements using a balance (天平) that allows you to compare the weight of 2 elements.

Minimize (使最小) the number of comparisons.

Max element

Find element with max weight (重量) from w[0, n-1]

```
maxElement=0
for (int i = 1; i < n; i++)
    if (w[maxElement] < w[i]) maxElement = i;
```

Number of comparisons (比较次数) is n–1.

Obvious method

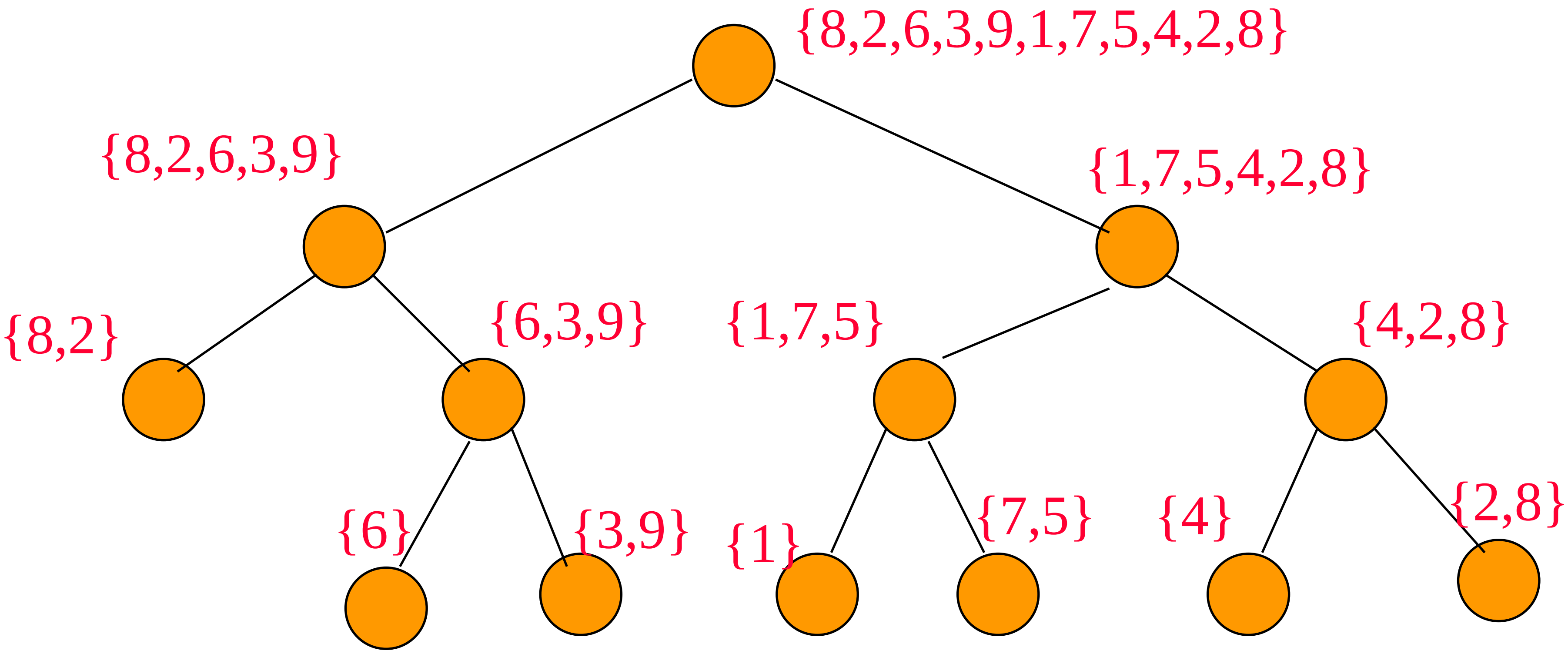
- Find the max of n elements making $n-1$ comparisons.
- Find the min of the remaining $n-1$ elements making $n-2$ comparisons.
- Total number of comparisons is $2n-3$.

Divide and Conquer

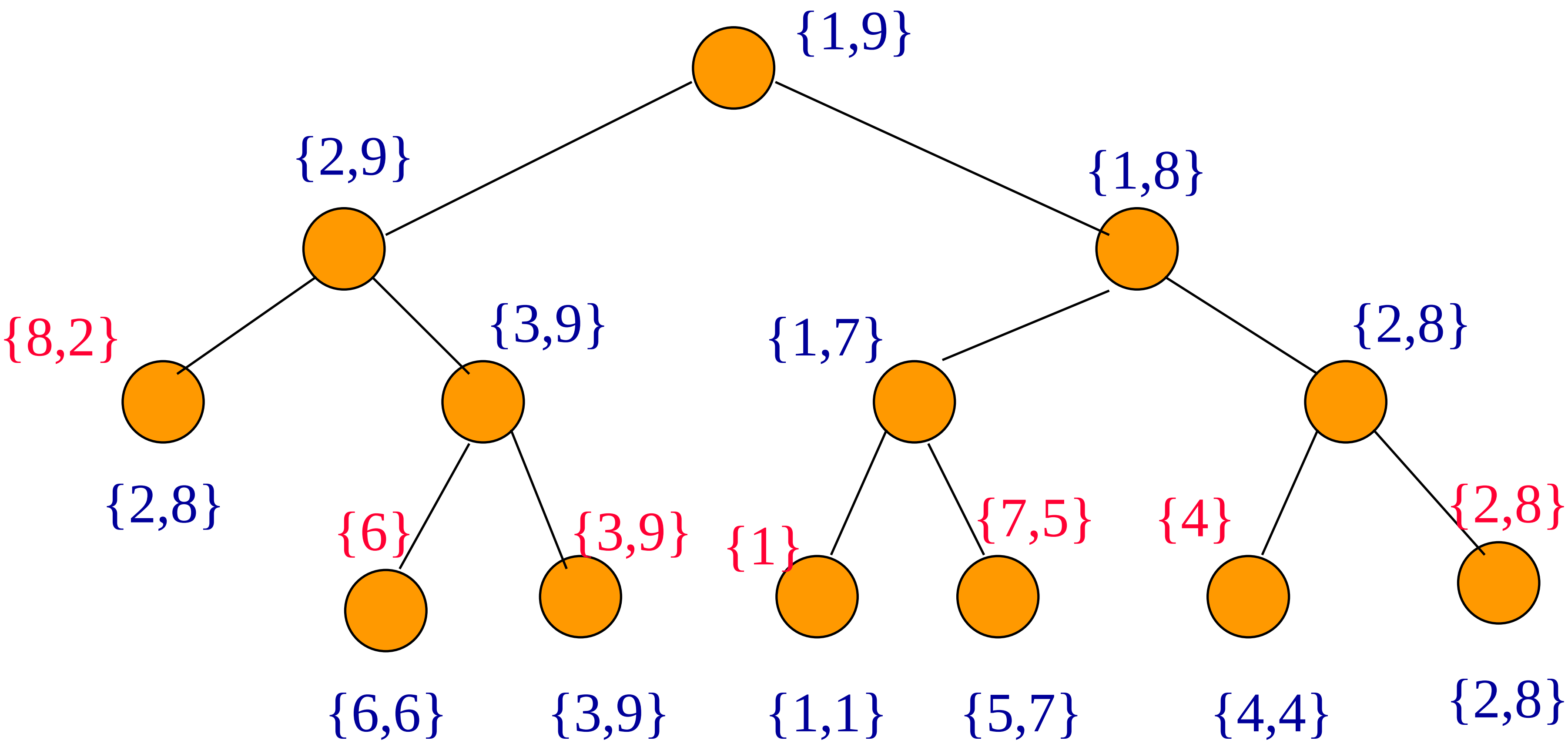
How to combine?

- Find the min and max of {3,5,6,2,4,9,3,1}.
- Large instance.
- A = {3,5,6,2} and B = {4,9,3,1}.
- min(A) = 2, min(B) = 1.
- max(A) = 6, max(B) = 9.
- min{min(A),min(B)} = 1.
- max{max(A), max(B)} = 9.

Dividing Into Smaller Problems



Solve Small Problems and Combine



MaxMin(L)

```
{
    if length(L)=1 or 2, we use at most one comparison.
    else {
        split (分裂) L into lists L1 and L2, each of n/2 elements
        (min1, max1) = MaxMin(L1)
        (min2, max2) = MaxMin(L2)
        return (Min(min1, min2), Max(max1, max2))
        }
 }
```

Complexity analysis (Number of Comparisons)

$$T(1)=0,\ T(2)=1,$$

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

Assume $n=2^k$, we have

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

$$2T(n/2)= 4T(n/4)+2^2$$

$$\ldots.$$

$$2^{k-2}T(4)=2^{k-1}T(2)+2^{k-1}$$

$$T(n)= 2^{k-1}+2+2^2+\ldots+2^{k-1}=3\times 2^{k-1}-2=3n/2-2.$$

Time	$2n–3$	$3n/2–2$
1 minute	n=31	n=41
1 hour	n=1801	n=2401
1 day	n=43201	n=57601

Assume that one comparison takes one second.

Interpretation (解释) of Divide-and-Conquer

- The working of a divide-and-conquer algorithm can be described (描述) by a tree -- recursion tree.
- The algorithm moves down the recursion tree dividing large problems into smaller ones.
- Leaves (叶子) represent small problems.
- The algorithm moves back up the tree combining the results from the subtrees (子树).

A general divide-and-conquer algorithm

- **Step 1**: If the problem size is small, solve this problem directly; otherwise, split the original problem into b sub-problems of size n/a.
- **Step 2**: Recursively solve these b sub-problems by applying (应用) this algorithm.
- **Step 3**: Merge the solutions of the b sub-problems into a solution of the original problem.

Time complexity of the general algorithm

- Time complexity:

$$\mathrm{T}(n)=\begin{cases} b\mathrm{T}(n/a)+\mathrm{S}(n)+\mathrm{M}(n) & n\geq c \\ d & n<c \end{cases}$$

where, S(n) : time for splitting

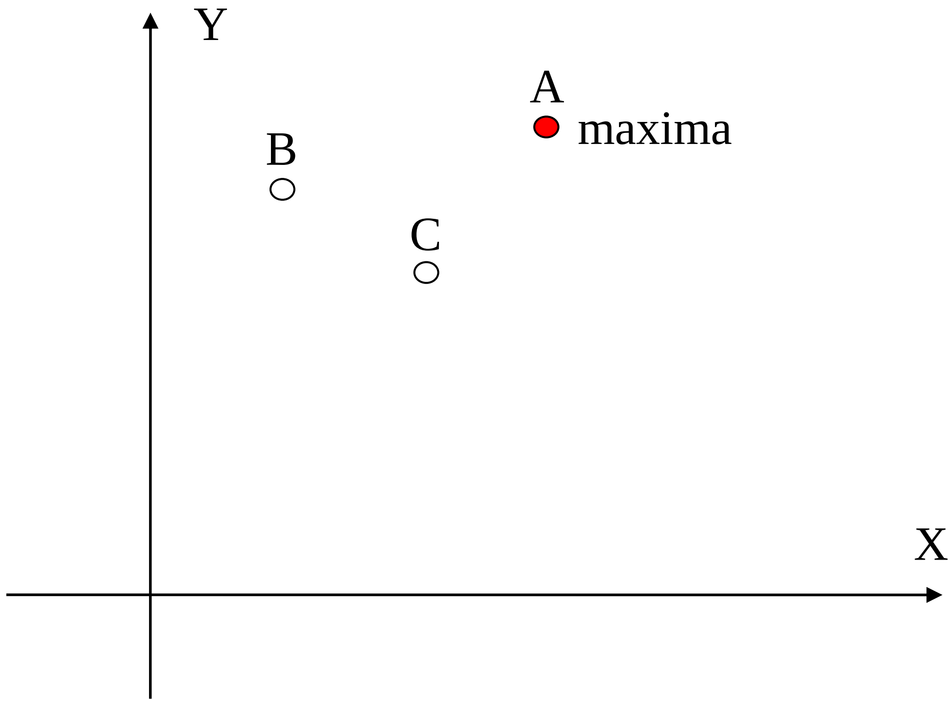
M(n) : time for merging

b, c and d are constants

- e.g. Merge-Sort, FastMutiply and MaxMin.

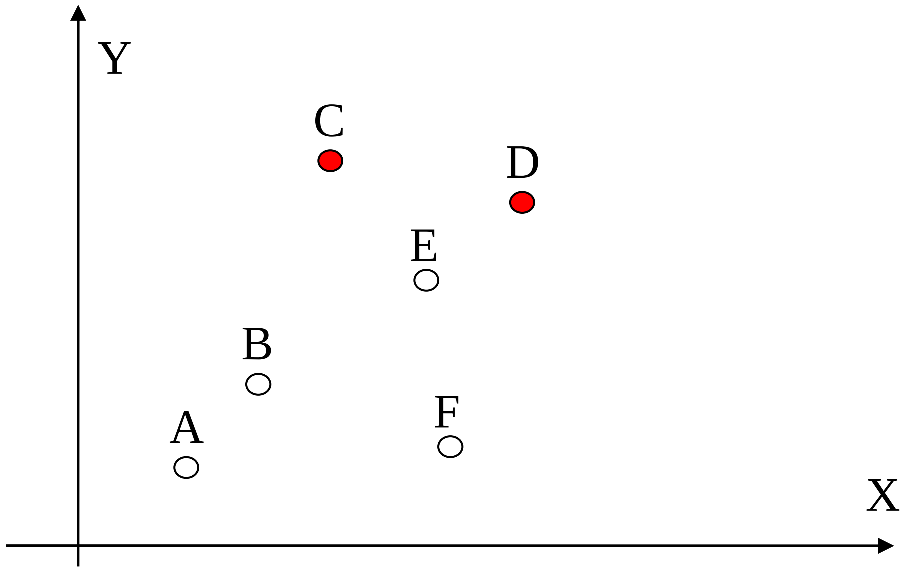
Example 5. 2-D (二维) maxima (极大点) finding problem

Definition (定义)：A point (x_1, y_1) *dominates* (支配) (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$. A point is called a *maxima* if no other point dominates it.



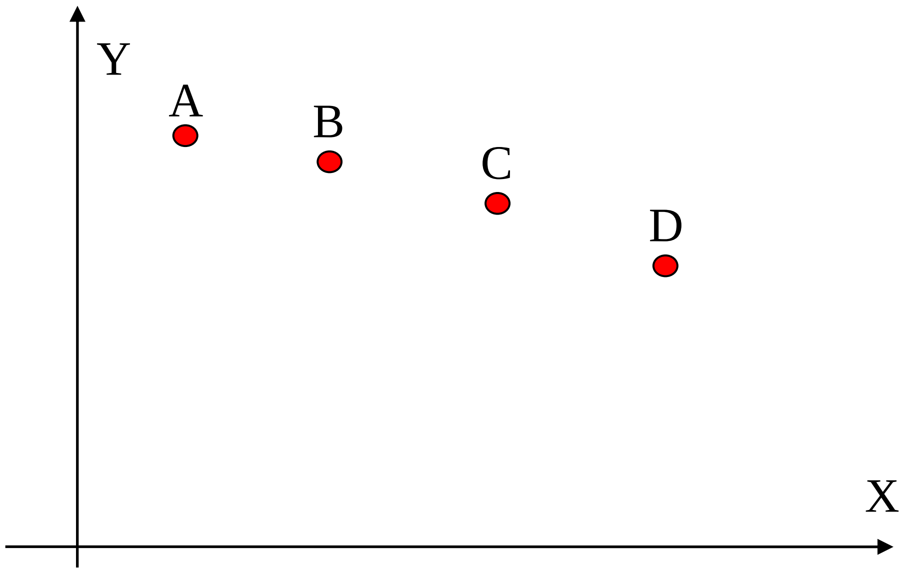
Solution 1 :

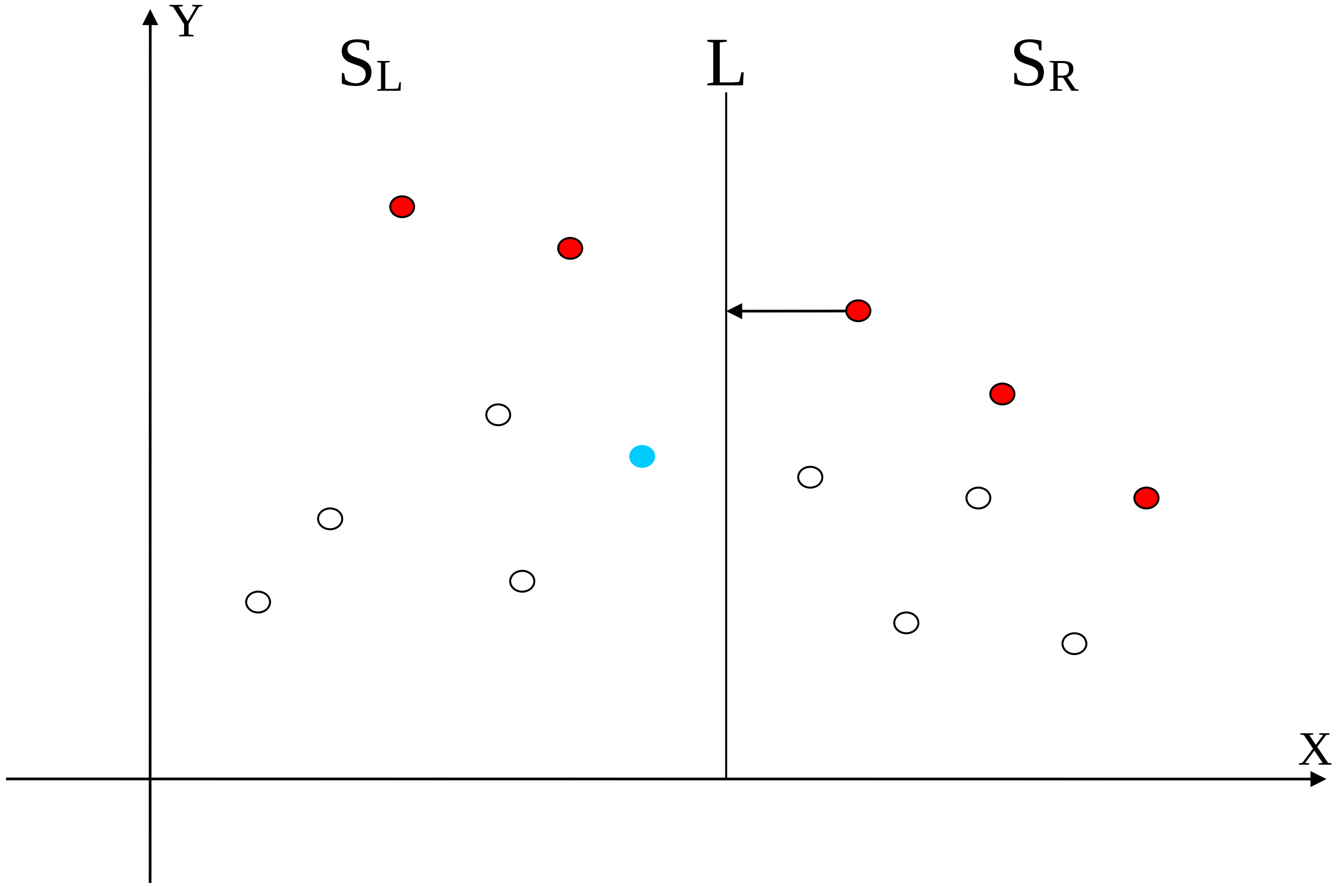
- Compare every pair of points and take it as a competition.
- If point A dominates point B, then A wins the match and B is kicked out.



Complexity analysis: $O(n^2)$.

The worst case occurs if all the points are maximal points.




Y
S_L
L
S_R
X

Solution 2 (Divide and Conquer) :

- ■ Input: A set of *n* planar (平面) points.
- ■ Output: The *maximal* points of S.

Step 1: If S contains only one point, return it as the *maxima*. Otherwise, find a line L perpendicular (垂直) to the X-axis which separates the set of points into two subsets (子集) S_L and S_R, each of which consisting of *n*/2 points.

Step 2:   Recursively (递归) find the *maximal* points of S_L and S_R.

Step 3: Find the largest y-value of S_R. Project (投影) the *maximal* points of S_L onto L. Discard (丢弃) each of the maximal points of S_L if its y-value is less than the largest y-value of S_R.

Time complexity analysis: $T(n)$

Step 1: At most $O(n\log n)$. In fact, this can be done in $O(n)$.

Step 2: $2T(n/2)$

Step 3: $O(n)$

So, we have

$$T(n)=2T(n/2)+O(n).$$

Applying (应用) Master Theorem, we have

$$T(n)=O(n\log n).$$

If

$$T(n)=2T(n/2)+O(n\log n),$$

then

$$T(n)=O(n\log^2 n)$$

Example 6. Majority (多数) Problem

Problem

Given an array (数组) A of n elements, use "=" test only to find the *majority element* (which appears more than $n/2$ times) in A.

For example, given (2, 3, 2, 1, 3, 2, 2), then 2 is the majority element because 4>7/2.

- Trivial (平凡) solution: counting (计数) is $O(n^2)$.
- Can't apply any sort algorithm, since only "=" test is allowed.

Divide and Conquer

Majority(A[1, *n*])
if *n*=1, then
    return A[1]
else
    m1=Majority(A[1, *n*/2])
    m2=Majority(A[*n*/2+1, *n*])
test if m1 or m2 is the majority for A[1, *n*]
return majority or no majority.

– Complexity analysis

■ This algorithm uses $T(n) = 2T(n/2)+O(n) = O(n\log n)$ time.

– However, there is a linear time algorithm for the problem.

– Moral (寓意) of the story

  - ■ Divide and conquer may not always give you the best solution.

– Remember

  - ■ Divide-and-conquer works if the interaction (交互) between subproblems is relatively simple. Usually the subproblems do not overlap (重叠).

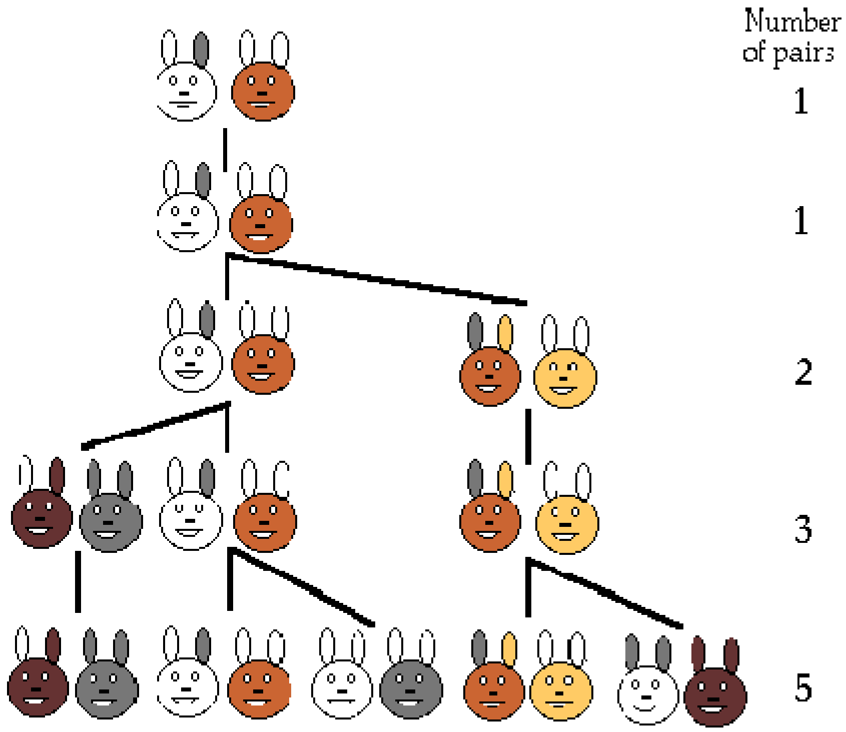
Paradigm 3:
Dynamic (动态) Programming (规划)

Example 1. Fibonacci numbers

Originally (最初) in the year 1202, Fibonacci was presented (提出) with a problem of how quickly the rabbit (兔子) population will grow in ideal (理想) conditions:

We assume (假定):

- ✓ We have two newly-born rabbits (1 male, 1 female)
- ✓ Rabbits produce another pair of newly born rabbits once a month after they are 2 months old
- ✓ They always give birth to twins (1 male, 1 female)
- ✓ They never die and never stop propagating (繁殖)

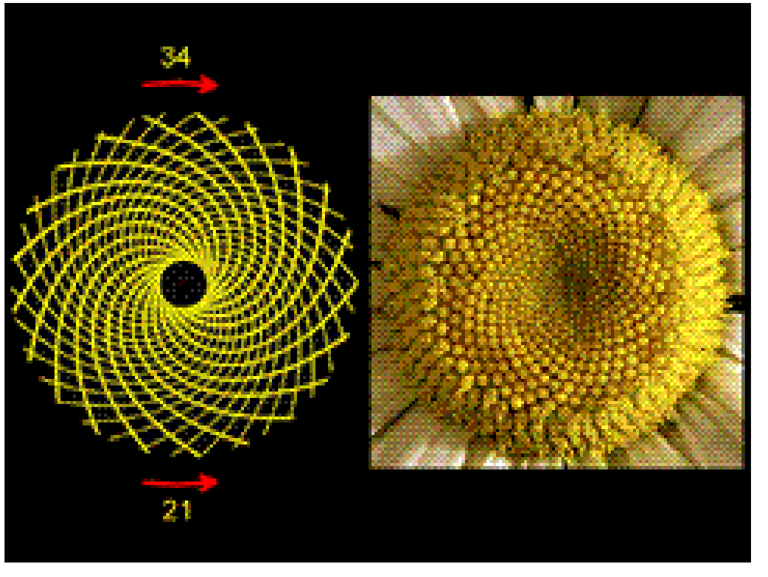

Number
of pairs
1
1
2
3
5

Defined by Recursion

$$F_0 = 0 \qquad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

$F_2=1$; $F_3=2$; $F_4=3$; $F_5=5$; $F_6=8$; $F_7=13$; $F_8=21$; $F_9=34$; $F_{10}=55$; $F_{11}=89$ ......



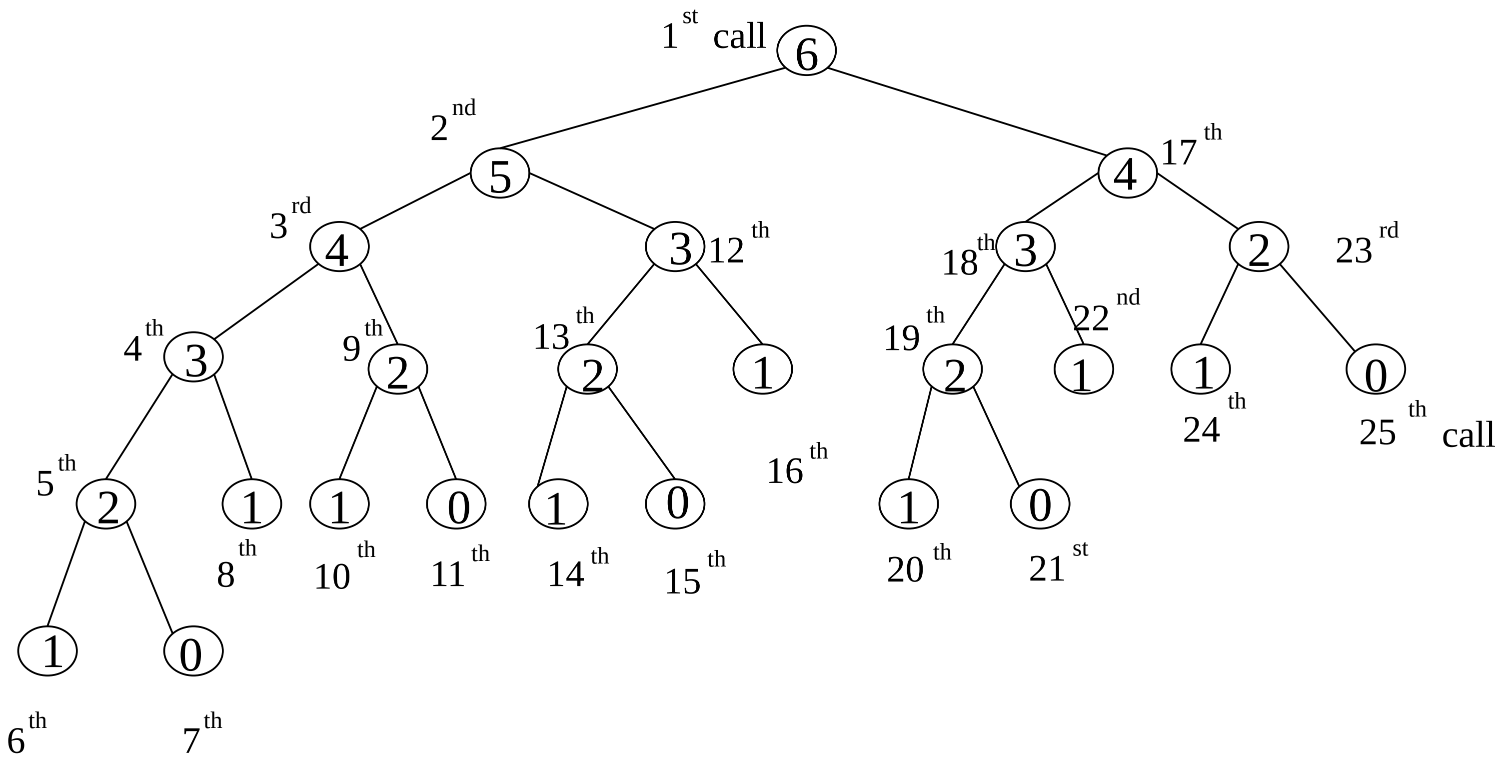
Problem: computing F(n)

A naive algorithm

```
F(n) {
    if (n<2) return n;
    else return F(n-1)+F(n-2);
}
```

Very inefficient (效率低): how many calls ?

For example: for F(6) ?


1st call
6
2nd
5
17th
4
3rd
4
3
12th
18th
3
2
23rd
4th
3
9th
2
13th
2
1
19th
2
22nd
1
1
0
24th
25th call
5th
2
1
1
0
1
0
16th
1
0
8th
10th
11th
14th
15th
20th
21st
1
0
6th
7th

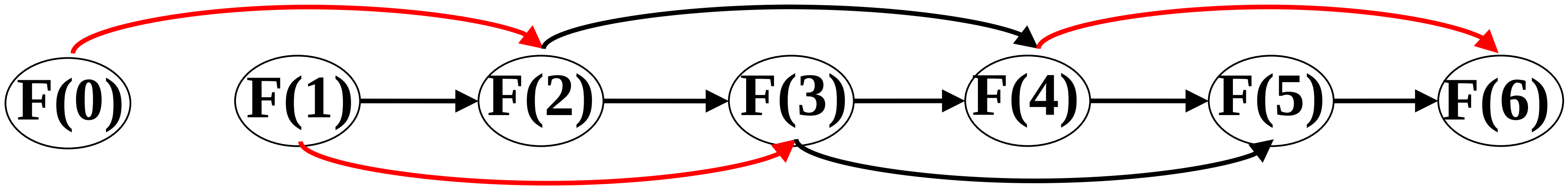
Complexity analysis:

The number of recursive calls (递归调用) $T(n)$ needed to compute $F(n)$ is

$$T(0)=1$$
$$T(1)=1$$
$$T(n)=T(n-1)+T(n-2)+1 \text{ for } n\geq 2.$$

It can be proved that $T(n)=\Omega(1.6^n)$. So the running time is exponential.

Reverse (反向) graphical order – dynamic programming



Observation (观察): Any subproblem only depends on its two predecessor (前辈) subproblems.

F_0	F_1	F_2	F_3	F_4	F_5	F_6
0	1	1	2	3	5	8

Dynamic Programming Algorithm

```
F(n) {
    curr=n, ppred=0, pred=1;
    for (j=2; j<n+1; j++) {
        curr=pred+ppred;
        ppred=pred;
        pred=curr; }
    return curr;
}
```

Complexity analysis: $T(n)= \Theta(n)$. So, the running time is linear (线性的).

What is dynamic programming?

- Main original (原始) motivation (动机):
  - ✓ replace an exponential-time computation by a polynomial-time computation
- Splitting problems into sub-problems may be very expensive.
  - ✓ if not controlled correctly, many subproblems will be solved repeatedly
- Dynamic programming:
  - ✓ Basic idea: store (存储) results for subproblems rather than re-computing (重新计算) them.
  - ✓ Applicable (适用于) to problems where a recursive algorithm solves many subproblems repeatedly.

Homework 5

令 A[1..n]是一个由 n 个数所组成的数组。序列 A[1], A[2], … , A[n]被称为是单模的(unimodal)，当且仅当存在**顶点序号** $1 \le p \le n$，使得数组的元素从 A[1]、A[2]开始到 A[p]单调增加，而从 A[p]、A[p+1]开始到 A[n]则单调下降。问题：对于一个给定的单模序列 A[1], A[2], … , A[n]，请找出其**顶点序号** p。设计一个求解此问题的算法并分析其最坏时间复杂性。

Homework 6

设有一个算法Median能在$O(n)$的时间内计算一个数组的中位值(即将数组的元素按大小顺序排列正好位于中间的值)。给定一个有n个元素的数组，能否以Median算法为基础设计一个算法，对任意的整数$1 \le i \le n$，该算法在$O(n)$的时间内求出数组中第i大小的元素。如果能，请给出一个这样的算法并分析其最坏时间复杂性。