Analysis and Design of Algorithms

Chapter 5: Decrease and Conquer

School of Software Engineering © Yaping Zhu

Decrease and Conquer

- ***Decrease and Conquer tech.***

  Exploit the relationship between a solution to a given instance of a problem and a solution to a smaller instance（利用一个问题给定实例的解和同样问题较小实例的解之间的某种关系）

  Once the relationship is established, the problem can be exploited either top down (自底向上，非递归) or bottom up (自顶向下，递归).

  Three major variations of decrease-and-conquer:

  - Decrease by a constant --减常量
  - Decrease by a constant factor --减常因子
  - Variable size decrease --减可变规模

Decrease and Conquer

Three variations of Decrease and Conquer tech.

1. Decrease by a constant

The size of the problem is reduced by the same constant on each iteration/ recursion of the algorithm (每次迭代从实例中减去一个相同常量)

e.g.

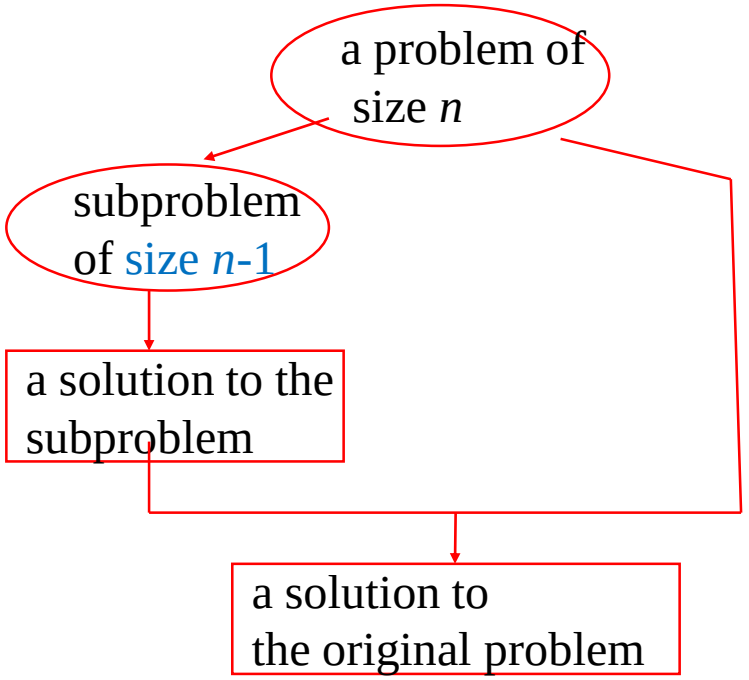
- computing a^n

规模为*n*的实例与规模为*n*-1的实例之间的关系：

$$a^n = a * a^{n-1}$$

f(n)=a^n 的递归定义（自顶向下）：

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



Decrease and Conquer

Three variations of Decrease and Conquer tech.

2. Decrease by a constant factor (by half)

The size of the problem is reduced by the same constant factor on each iteration/recursion of the algorithm.

e.g.

- computing a^n

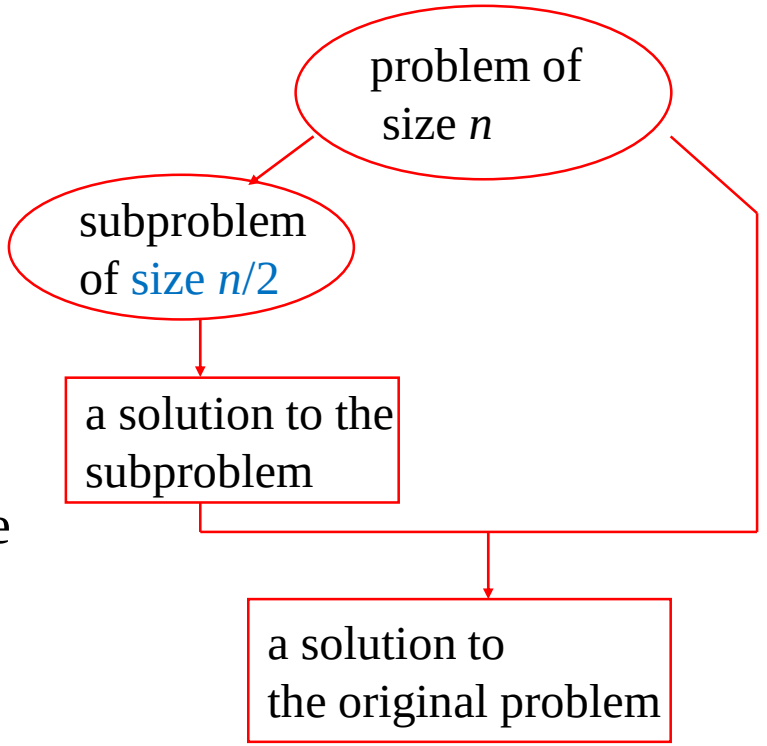
规模为n的实例与规模减半的实例之间的关系：

$$a^n=(a^{n/2})^2$$

递归算法：

$$a^n=\begin{cases}(a^{n/2})^2 & \text{if } n \text{ is even and positive}\\ (a^{(n-1)/2})^2\cdot a & \text{if } n \text{ is odd and } n>1\\ a & \text{if } n=1\end{cases}$$

算法效率：Θ(log n)

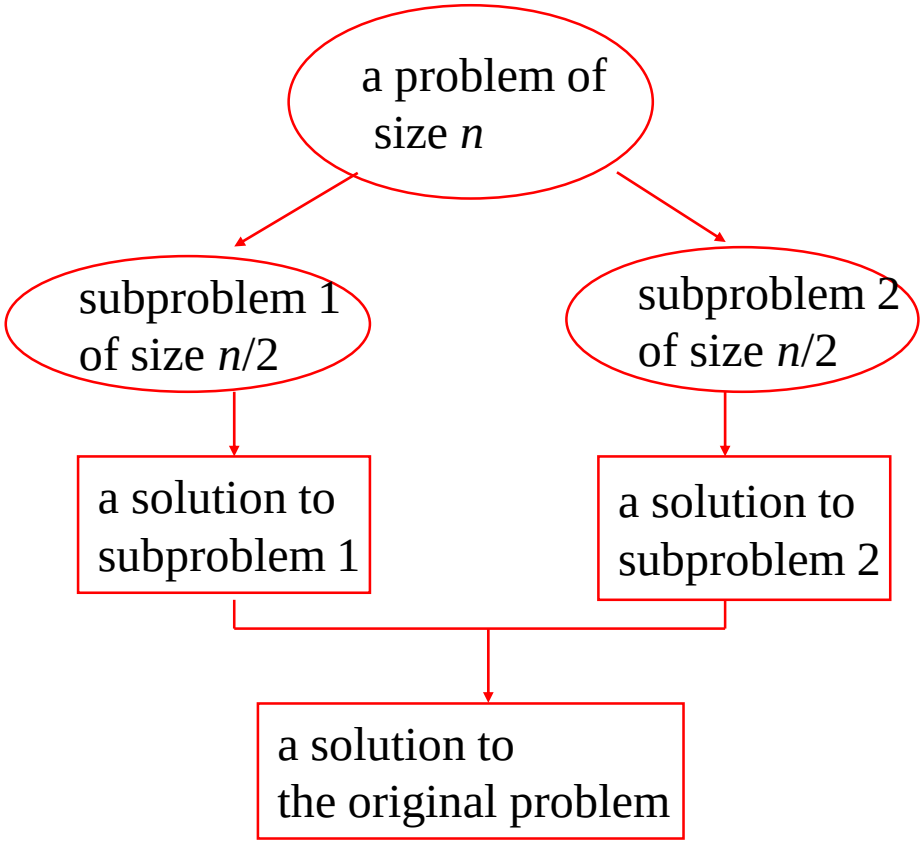


Decrease and Conquer

- ***Three variations of Decrease and Conquer tech.***

  For comparison: Divide and Conquer (later)

$$a^n = \begin{cases} a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil} & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$



Decrease and Conquer

- ***Three variations of Decrease and Conquer tech.***

  3. Variable-size decrease

  The size reduction pattern varies from one iteration of an algorithm to another (每次迭代规模减小的模式不同)

  e. g.

  - Euclid's algorithm

$$\gcd(m, n) = \gcd(n, m \bmod n) \quad \text{iteratively while } n \neq 0$$
$$\gcd(m, 0) = m$$

e.g., gcd(60,24) = gcd(24,12) = gcd(12,0) =12

Decrease by One

- ***Insertion Sort algorithm***
- ***Topological Sorting Algorithm***

Insertion Sort

Insertion Sort algorithm

- *idea*
  - *Assume that the smaller problem of sorting array A*[0…*n*-2] *has been solved to give a sorted array of size n*-1: *A*[0…*n*-2].
  - *We can take advantage of this solution to the smaller problem to get a solution to the original problem, -------- to find an appropriate position for A*[*n*-1*] among the sorted elements A*[0…*n*-2] *and insert it there*.

Insertion Sort

Insertion Sort algorithm

- *Three ways to achieve it:*

- *scan the sorted subarray from left to right,*
  ----- the first element >= A[n-1]
  ------ insert A[n-1] before the element — **Insertion Sort**

- *scan the sorted subarray from right to left,*
  ----- the first element <= A[n-1]
  ------ insert A[n-1] after the element — **Insertion Sort**

- *use binary search to find appropriate position for A[n-1] in the sorted subarray* — **Binary Insertion Sort**

Insertion Sort

- ***Example***

1:	89|	**45**	68	90	29	34	17
2:	45	89|	**68**	90	29	34	17
3:	45	68	89|	**90**	29	34	17
4:	45	68	89	90|	**29**	34	17
5:	29	45	68	89	90|	**34**	17
6:	29	34	45	68	89	90|	**17**
7:	17	29	34	45	68	89	90

- The vertical bar separates the sorted part of the array from the remaining elements.
- The element to be inserted is in bold.

Insertion Sort

Insertion Sort: an Iterative Solution

ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Basic operation

$$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \mid A[i] \cdots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Insertion Sort

Analysis of Insertion Sort

Worst-case:

- $A[j] > v$ is executed for every $j = i-1,..0$
- If and only if $A[j] > A[i]$ for all $j = i-1,..0$
- i.e., the original input is an array of strictly decreasing values

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \theta(n^2)$$

Insertion Sort

Analysis of Insertion Sort

Best-case:

- $A[j] > v$ is executed only once on every iteration
- If and only if $A[i-1] <= A[i]$ for every $i = 1,..n-1$
- the original input is already sorted in ascending order

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \theta(n)$$

- *Application: almost sorted files*

Insertion Sort

Analysis of Insertion Sort

- *Average-case (Exer. 11):*

$$C_{avg}(n) \approx \frac{n^2}{4} \in \theta(n^2)$$

☆ 平均性能比最差性能快一倍

☆ Selection Sort： $\Theta(n^2)$； Bubble Sort： $\Theta(n^2)$；

☆ *Shell Sort* –对较大文件进行排序

While sorting by quicksort, after subarrays become smaller than some predefined size, we can switch to using insertion sort.

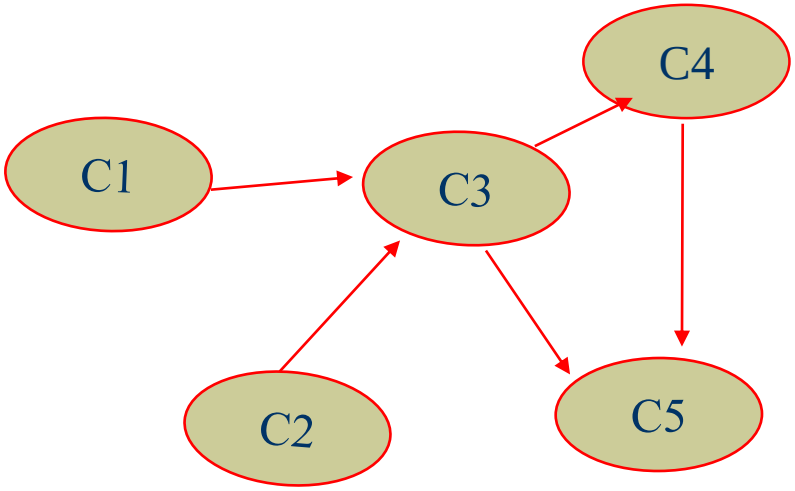
-- typically decreases the total running time of quicksort by about 10%

Topological Sorting

Problem

- *Example:* *Give an order of the courses so that the prerequisites are met.*

五门必修课的一个集合{C1, C2, C3, C4, C5}，一个在校的学生必须在某个阶段修完这几门课程。可以按照任何次序学习这些课程，只要满足下面这些先决条件：C1 和 C2 没有任何先决条件，修完 C1 和 C2 才能修 C3，修完 C3 才能修 C4，而修完 C3 和 C4 才能修 C5。这个学生每个学期只能修一门课程。这个学生应该按照什么顺序来学习这些课程？



Topological Sorting

Problem

- *The situation can be modeled by a diagraph:*
  - *vertices represent the courses*
  - *edges indicate the prerequisites requirements*

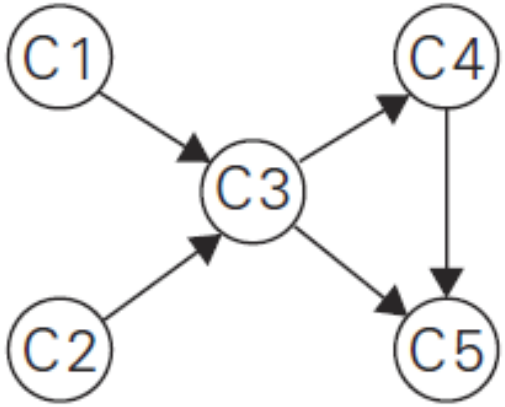
有向图和无向图的两个显著差异：

- 有向图的邻接矩阵不一定表现出对称性
- 有向图的一条边在图的邻接链表中只有一个（不是两个）相应节点

The question is how to list the vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

Topological Sorting

Problem



Is the problem solvable if the digraph contains a cycle?

- *Given a Directed Acyclic Graph (DAG, 有向无环图) G = (V, E), find a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.* --Topological Sorting
  - Being a DAG is not only necessary but also sufficient for Topological Sorting to be possible.

Topological Sorting

Topological Sorting

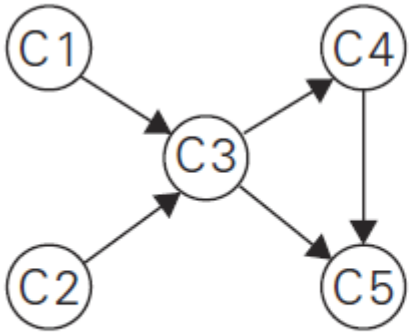
- There are two efficient algorithms that both verify whether a digraph is a DAG and, if it is, produce an ordering of vertices that solves the topological sorting problem.

  - ***DFS-Based method (深度优先查找)***
  - ***Source Removal method (基于减治技术)***

Topological Sorting

1. DFS-Based method (深度优先查找)

- *DFS traversal noting the order in which vertices are popped off stack (the order in which the dead end vertices appear)*
- *Reverse the above order*

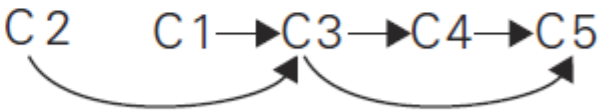


(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:



(c)

Topological Sorting

1. DFS-Based method (深度优先查找)

- *Questions：*
  - *Can we use the order in which vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?*
  - *Does it matter from which node we start?*

复杂度分析

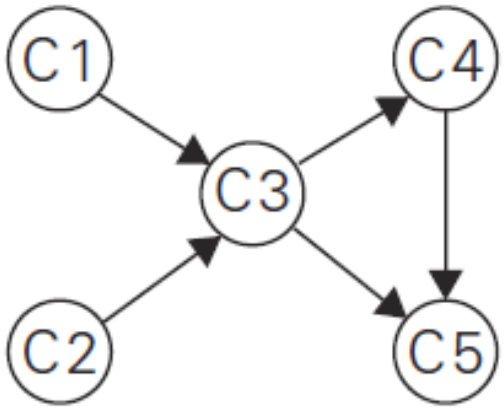
复杂度同DFS一致，即O(E+V)。

具体而言，首先需要保证图是有向无环图，判断图是DAG可以使用基于DFS的算法，复杂度为O(E+V)，而后面的拓扑排序也是依赖于DFS，复杂度为O(E+V)。

Topological Sorting

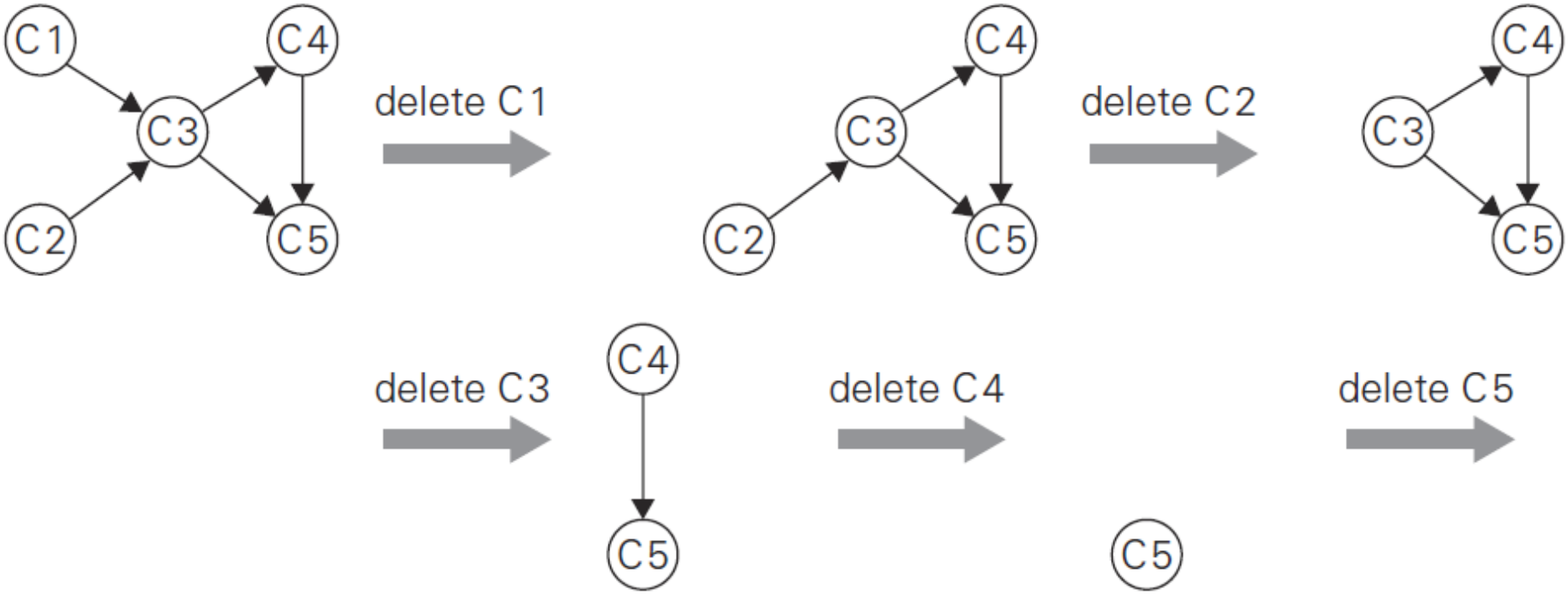
2. Source Removal method

- *Based on a direct implementation of the decrease-by-one techniques.*
- *Repeatedly identify and remove a source vertex, i.e., a vertex that has no incoming edges, and delete it along with all the edges outgoing from it.*



Topological Sorting

2. Source Removal method



Solution: C1, C2, C3, C4, C5

- Topological Sorting problem may have several alternative solutions.

复杂度分析

初始化入度为0的集合需要遍历整张图，检查每个节点和每条边，因此复杂度为O(E+V)；

然后对该集合进行操作，又需要遍历整张图中的每条边，复杂度也为O(E+V)；

故总体算法复杂度为O(E+V).

Topological Sorting

Applications:

- 程序编译中的指令调度
- 电子表格单元格的公式求值顺序
- 解决链接器中的符号依赖问题

Time efficiency for decrease-by-one

减一算法时间效率分析的递推方程的一般形式为：

$$T(n) = T(n-1) + f(n)$$

$f(n)$ 表示把一个实例化简为一个更小的实例并把更小实例的解拓展为更大实例的解所需要的时间。

使用反向替换法求解，得到：

$$\begin{aligned} T(n) &= T(n-1) + f(n) \\ &= T(n-2) + f(n-1) + f(n) \\ &= \cdots \\ &= T(0) + \sum_{j=1}^{n} f(j). \end{aligned}$$

Decrease-by-a-Constant-Factor

- Binary Search
- Fake-Coin Problem
- Russian Peasant Multiplication

Binary Search

- **Binary Search – a Recursive Algorithm (for sorted array)**

Compare a search key K with the array's middle element A[m]:

- If match, the algorithm stops;
- If K<A[m], repeat the operation for the first half of the array;
- If K>A[m], repeat the operation for the second half

$$K \updownarrow \underbrace{A[0] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\text{search here if } K>A[m]}.$$

Binary Search

Binary Search – a Recursive Algorithm

Example:

K=70, 对下面数组进行折半查找

3	14	27	31	39	42	55	70	74	81	85	93	98

迭代查找过程:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l*,*m*	*r*				

Binary Search

Binary Search – a Recursive Algorithm

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//        or −1 if there is no such element
l ← 0;    r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Basic operation:

while循环中k与A 中元素的比较运算

three-way comparison

Binary Search

Analysis of Binary Search

- *Basic operation:* *key comparison* (*three-way comparison*)
- *Worst-case* (*successful or fail*) *:*

$$\begin{cases} C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1, \\ C_{worst}(1) = 1 \end{cases}$$

  - Solution:

$C_{worst}(n) = \Theta(\log n)$

- *Best-case:*
  - successful $C_{best}(n) = 1$

一次找到，即$K=A[n/2]$

for $n = 2^k$,

$C(2^k) = C(2^{k-1}) + 1$ for $k>0$

$C(2^0) = 1$

backward substitutions:

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

$= [C(2^{k-2}) + 1] + 1$

$= C(2^{k-2}) + 2 = \ldots = C(2^{k-i}) + i \ \ldots$

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

then, $C(n) = \log_2 n + 1$

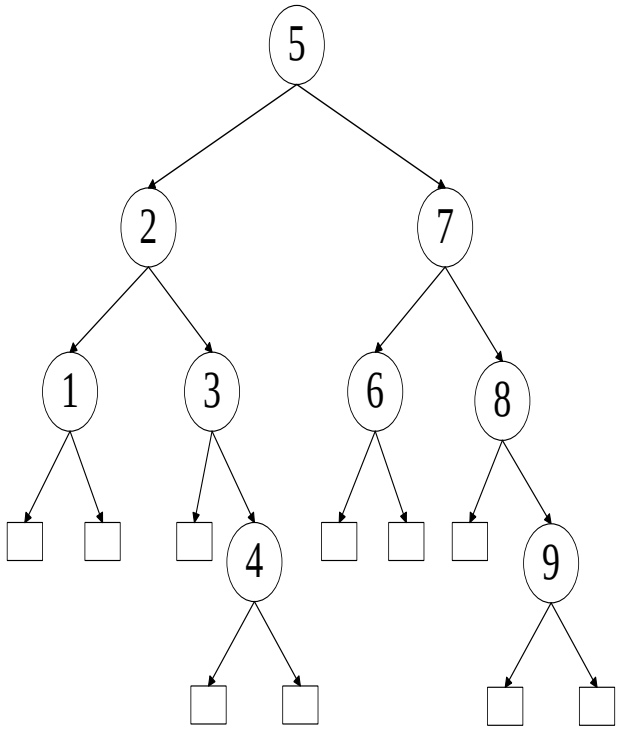
Binary Search

Analysis of Binary Search

Average-case:

- Consider the searching process as a binary tree. In looking at the binary tree, we see that there are i comparisons needed to search 2^{i-1} elements on level i of the tree.
- For a list with $n = 2^k-1$ elements, there are k levels in the binary tree.
- The average case for all successful search:

$$A(n) = \frac{1}{n}\sum_{i=1}^{k} i2^{i-1} \approx \log(n+1)-1$$



Fake-Coin Problem (假币问题)

Problem: Among ***n*** identical-looking coins, one is fake. With a balance scale (天平), we can compare any two sets of coins.

That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much.

The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here — assumes that the fake coin is known to be, say, lighter than the genuine one.

Fake-Coin Problem

Idea 1: Decrease by half

Step1: To divide ***n*** coins into two piles of ***n/***2 coins each, leaving one extra coin aside if ***n*** is odd, and put the two piles on the scale.

Step2: If the piles weigh the same, the coin put aside must be fake;

Step3: Otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

Fake-Coin Problem

Complexity:

worst case：

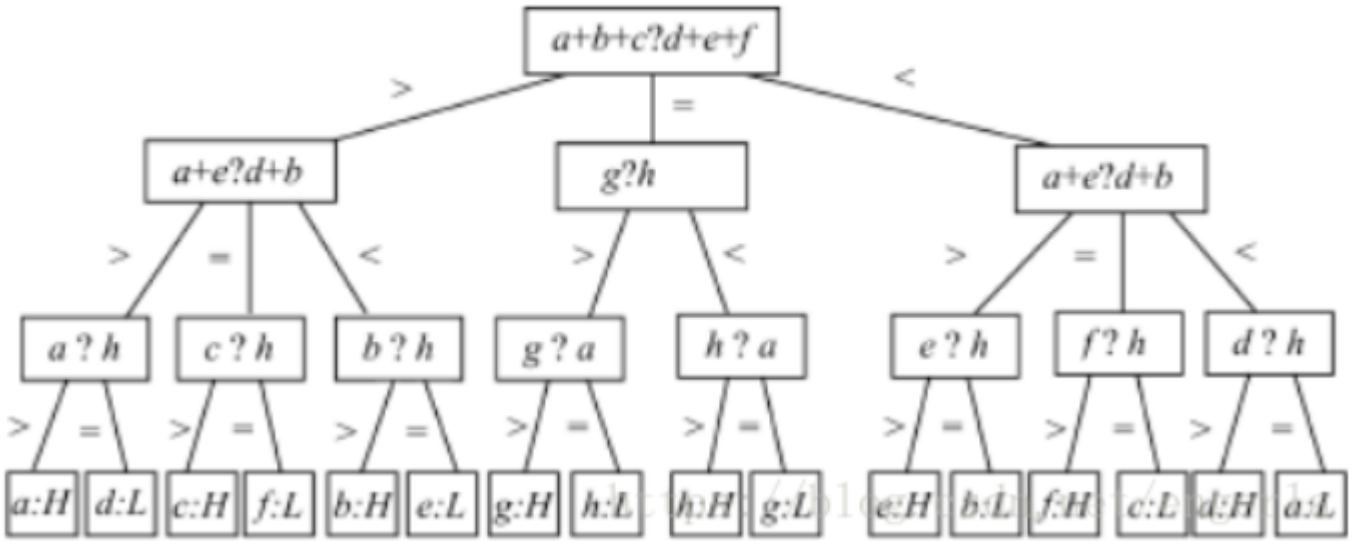
$$W(n) = W(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad W(1) = 0.$$

$W(n) = O(\log_2 n)$

Fake-Coin Problem

Idea 2: Decrease by a third

- **Step1**: To divide ***n*** coins into three piles. The first two piles with ***n***/3 coins each, and all the left as the third pile. Put the two piles on the scale.
- **Step2**: If the two piles weigh the same, the fake coin must be in the third pile;
- **Step3:** Otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.



Fake-Coin Problem

Complexity:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n/3) + 1 & n > 1 \end{cases}$$

$$T(n) = O(\log_3 n)$$

Fake-Coin Problem

```
1  const int N=8; //假设求解8枚硬币问题
2  int a[N]={2,2,1,2,2,2,2,2};//真币的重量是2，假币的重量是1
3  int Coin(int low,int high,int n)//在a[low]-a[high]中查找假币
4  {
5      int i,num1,num1,num3;//num1,num2和num3存储3组硬币的个数
6      int add1=0,add2=0;//add1和add2存储前两组硬币重量和
7      if(n==1)
8          return low+1;//返回序号，即下标+1
9      if(n%3 =0) //三组硬币的个数相同
10         num1=num2=n/3;
11     else
12         num1=num2=n/3+1;//前两组由[n/3]枚硬币
13     num3=n-num1-num2;
14     for(i=0;i<num1;i++) //计算第一组硬币的重量和
15         add1=add1+a[low+1];
16     for(i=num1;i<num1+num2;i++)
17         add2=add2+a[low+1];
18     if(add1<add2) //在第1组查找，下标范围是low-low+num1-1
19         return Coin(low,low+num1-1,num1);
20     else if (add1>add2)//在第2组查找，下标范围是low+num1到low+num1+num2-1
21         return Coin(low+num1,low+num1+num2-1,num2);
22     else  //在第1组查找，
23         return Coin(low+num1+num2,high,num3)'
24  }
```

Russian Peasant Multiplication (俄氏乘法)

Problem: Let ***n*** and ***m*** be positive integers whose product we want to compute, and let us measure the instance size by the value of ***n***. Now, if ***n*** is even, an instance of half the size has to deal with ***n/***2, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

If n is odd, we need only a slight adjustment of this formula:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

Using these formulas and the trivial case of $1 \cdot \boldsymbol{m} = \boldsymbol{m}$ to stop. We can compute product $\boldsymbol{n} \cdot \boldsymbol{m}$ either recursively or iteratively.

Russian Peasant Multiplication

An example of computing 50 * 65 with this algorithm:

$$n \cdot m = \frac{n}{2} \cdot 2m$$

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

What's the time complexity of this algorithm?

Time efficiency for decrease-by-a-constant-factor

减常因子算法时间效率分析的递推方程的一般形式为：

$$T(n) = T(n/b) + f(n) \qquad b>1$$

$f(n)$ 表示把一个实例化简为一个更小的实例并把更小实例的解拓展为更大实例的解所需要的时间。

基于平滑法则，使用反向替换法求解，得到：

$$\begin{aligned} T(b^k) &= T(b^{k-1}) + f(b^k) \\ &= T(b^{k-2}) + f(b^{k-1}) + f(b^k) \\ &= \cdots \\ &= T(1) + \sum_{j=1}^{k} f(b^j). \end{aligned}$$

Summary

- Decrease-and-conquer is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (usually recursively) or bottom up.

- There are three major variations of decrease-and-conquer:
  - decrease-by-a-constant: most often by one (e.g., insertion sort).
  - decrease-by-a-constant-factor: most often by the factor of two (e.g., binary search).
  - variable-size-decrease: (e.g., Euclid's algorithm)

Summary

- Insertion sort is a direct application of the decrease-(by one)-and-conquer technique to the sorting problem. It is a $\Theta(n^2)$ algorithm both in the worst and average cases, but it is about twice as fast on average than in the worst case. The algorithm's notable advantage is a good performance on almost-sorted arrays.

- A digraph is a graph with directions on its edges. The topological sorting problem asks to list vertices of a digraph in an order such that for every edge of the digraph, the vertex it starts at is listed before the vertex it points to. This problem has a solution if and only if a digraph is a dag (directed acyclic graph), i.e., it has no directed cycles.

- There are two algorithms for solving the topological sorting problem. The first one is based on depth-first search; the second is based on a direct application of the decrease-by-one technique.