

Analysis and Design of Algorithms

Chapter 6: Decrease and Conquer



School of Software Engineering © Ye Luo



Decrease and Conquer

■ Three variations of Decrease and Conquer tech.

exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance

1. Decrease by a constant

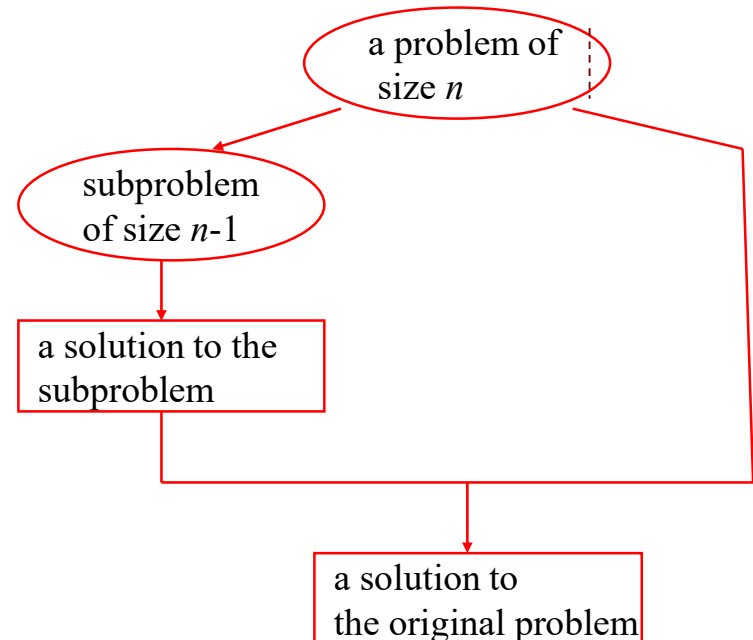
the size of the problem is reduced by the same constant on each iteration/
recursion of the algorithm.

Eg.

- $n !$

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

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



Decrease and Conquer

■ Three variations of Decrease and Conquer tech.

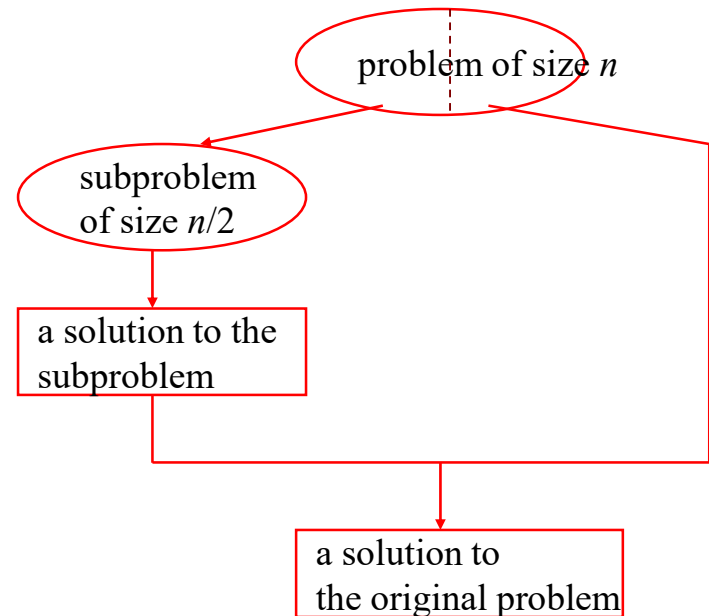
2. Decrease by a constant factor (by half)

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

Eg.

■ $a^n = (a^{n/2})^2$

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

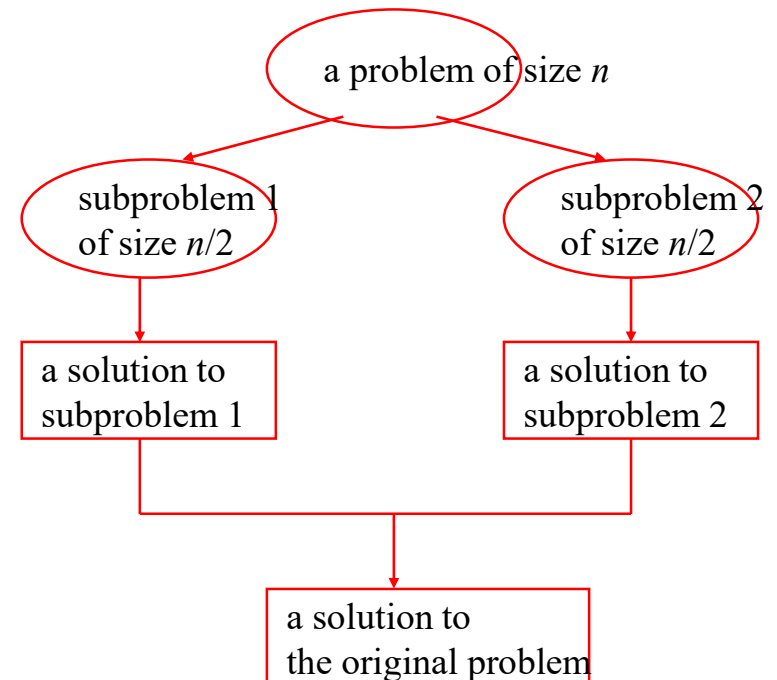


Decrease and Conquer

■ Three variations of Decrease and Conquer tech.

for comparison: Divide and Conquer

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



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.

Eg.

- *Euclid's algorithm*

$\gcd(m, n) = \gcd(n, m \bmod n)$ iteratively while $n \neq 0$

$\gcd(m, 0) = m$

Decrease by One

■ *Insertion Sort algorithm*

■ *Topological Sorting Algorithm*

Insertion Sort

■ Insertion Sort algorithm

✦ idea

- assume that the smaller problem of sorting array $A[0 \dots n-2]$ has been solved to give a sorted array of size $n-1$: $A[0 \dots n-2]$
- 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 $A[0 \dots n-2]$ and insert it there.

✦ Three ways to achieve it:

- scan the sorted subarray from left to right,
----- the first element $\geq A[n-1]$
----- insert $A[n-1]$ before the element *Insertion Sort*
- scan the sorted subarray from right to left,
----- the first element $\leq A[n-1]$
----- 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

■ Insertion Sort: an Iterative Solution


ALGORITHM InsertionSortIter(A[0..n-1])

//An iterative implementation of insertion sort

//Input: An array A[0..n-1] of n orderable elements

//Output: Array A[0..n-1] sorted in nondecreasing order

```
for  $i \leftarrow 1$  to  $n - 1$  do           // i: the index of the first element of the unsorted part.
     $v = A[i]$ 
     $j = i - 1$                        // j: the index of the sorted part of the array
    while  $j \geq 0$  and  $A[j] > v$  do //compare the key with each element in the sorted part
         $A[j+1] \leftarrow A[j]$ 
         $j \leftarrow j-1$ 
     $A[j+1] \leftarrow v$               //insert the key to the sorted part of the array
```



$A[0] < \dots < A[j] < A[j+1] < \dots < A[i-1] \mid \textcolor{red}{A[i]} \dots A[n-1]$
Smaller than or equal to $A[i]$ greater than $A[i]$

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

■ Analysis of Insertion Sort

✦ Worst-case:

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

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

Insertion Sort

■ Analysis of Insertion Sort

✦ Best-case:

- $A[j] > v$ is executed only once on every iteration
- If and only if $A[i-1] \leq A[i]$ for every $i = 1, \dots, n-1$
- 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*

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%

Insertion Sort

▣ *Analysis of Insertion Sort*

✦ *Average-case:*

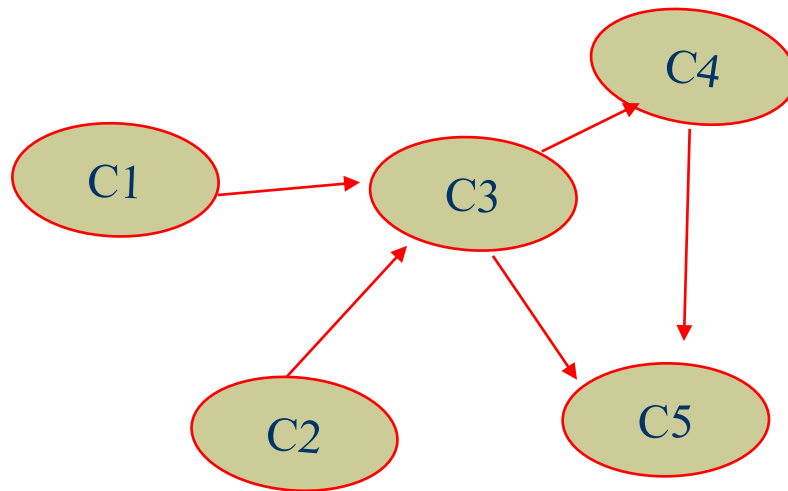
$$C_{avg}(n) \approx \frac{n^2}{4} \in \theta(n^2)$$

- ☆ Selection Sort $\Theta(n^2)$; Bubble Sort $\Theta(n^2)$;
- ☆ Shell Sort see Exer. 5.1 -10

Topological Sorting

❏ **problem**

- *Problem: Given a **Directed Acyclic Graph(DAG)** $G = (V, E)$, 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.*
- *Example: Give an order of the courses so that the prerequisites are met*



Directed graph (**digraph**):
All edges with directions;
Adjacency matrix not have to be symmetric

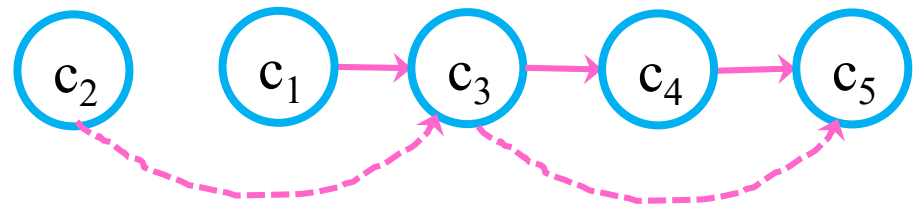
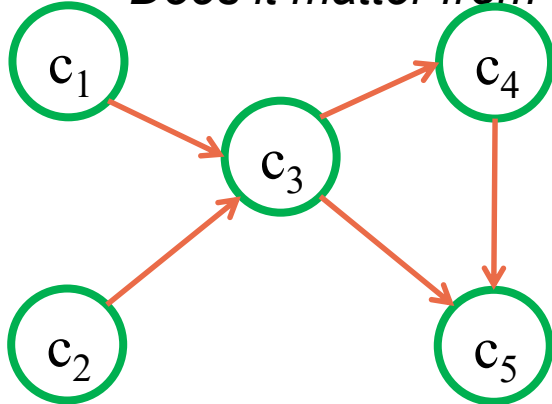
- *Being a dag is not only necessary but also sufficient for Topological Sorting to be possible*

Is the problem solvable if the digraph contains a cycle?

Topological Sorting

1. DFS-Based method

- 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
- 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

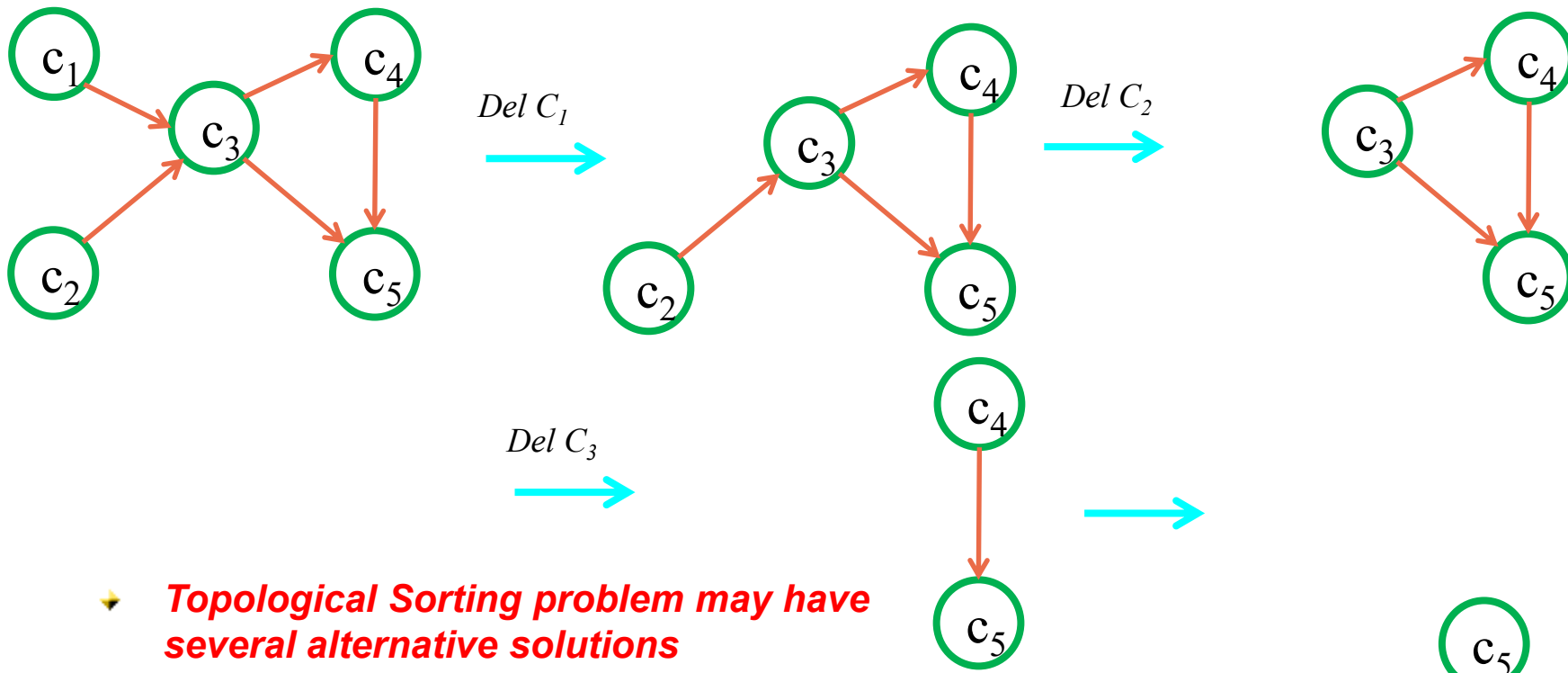
复杂度同DFS一致，即 $O(E+V)$ 。

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**, ie, a vertex that has no incoming edges, and delete it along with all the edges outgoing from it.



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 $O(E+V)$ ；

Decrease-by-a-Constant-Factor

■ *Binary Search*

■ *Fake-Coin Problem*

■ *Russian Peasant Multiplication*

Binary Search

■ Binary Search – a Recursive Algorithm

ALGORITHM BinarySearchRecur($A[0..n-1]$, l , r , K)

if $l > r$

 return -1

else

$m \leftarrow \lfloor (l + r) / 2 \rfloor$

 if $K = A[m]$

 return m

 else if $K < A[m]$

 return BinarySearchRecur($A[0..n-1]$, l , $m-1$, K)

 else

 return BinarySearchRecur($A[0..n-1]$, $m+1$, r , K)

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) :

$$\begin{aligned}C_w(n) &= C_w(\lfloor n/2 \rfloor) + 1, \\C_w(1) &= 1\end{aligned}$$

■ Solution:

$$C_w(n) = \Theta(\log n)$$

✦ *Best-case*:

■ successful $C_b(n) = 1$

一次找到, 即 $K = A[n/2]$

for $n = 2^k$,

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

$$C(2^0) = 1$$

backward substitutions:

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

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

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

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

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

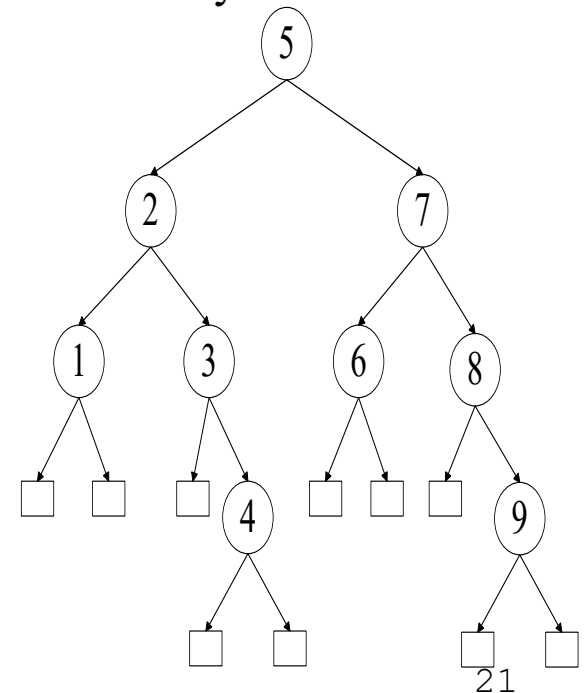
Binary Search

■ 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:

$$A(n) = \frac{1}{n} \sum_{i=1}^k i 2^{i-1} \approx \log(n+1) - 1$$



Fake-Coin Problem

Problem: Among n identical-looking coins, one is fake. With a balance scale, 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:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

算法的复杂度为 $O(\log_2 n)$ 。

Fake-Coin Problem

Idea 1: 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:

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

算法的复杂度为 $O(\log_3 n)$ 。

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,num2,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+i];
16     for(i=num1;i<num1+num2;i++)
17         add2=add2+a[low+1+i];
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3组查找,
23         return Coin(low+num1+num2,high,num3);
24 }
```

Russian Peasant Multiplication

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 $1 \cdot m = m$ to stop. We can compute product $n \cdot m$ either recursively or iteratively.

Russian Peasant Multiplication

An example of computing $50 \cdot 65$ with this algorithm

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

(a)

<i>n</i>	<i>m</i>	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		<u>3250</u>

(b)

What's the time complexity of this algorithm? Which one is an iterative algorithm?

Summary

- 减治法是一种一般性的算法设计技术，它利用了一个问题给定实例的解和同样问题较小实例的解之间的关系。一旦建立了这样一种关系，我们既可以自顶至下(递归)也可以自底至上地运用它(非递归)。
- 减治方法有 3 种主要的变种：
 - ◆ 减一个常量，常常是减一(例如插入排序)；
 - ◆ 减一个常因子，常常是减去因子 2(例如折半查找)；
 - ◆ 减可变规模(例如欧几里得算法)。
- 插入排序是减(减一)治技术在排序问题上的直接应用。无论在平均情况还是最差情况下，它都是一个 $\Theta(n^2)$ 的算法，但在平均情况下的效率大约要比最差情况快两倍。该算法一个较为出众的优势在于，对于几乎有序的数组，它的性能是很好的。
- 一个有向图是一个对边指定了方向的图。拓扑排序要求按照这种次序列出它的顶点，使得对于图中每一条边来说，边的起始顶点总是排在边的结束顶点之前。当且仅当有向图是一个无环有向图(不包含回路的有向图)的时候，该问题有解，也就是说，它不包含有向的回路。
- 解决拓扑排序问题有两种算法。第一种算法基于深度优先查找；第二种算法基于减一技术的直接应用。