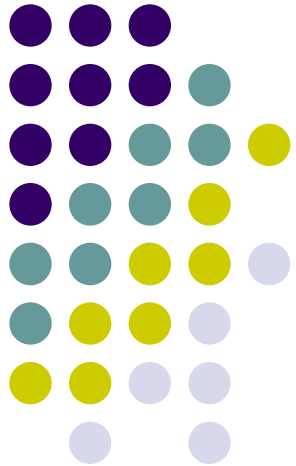
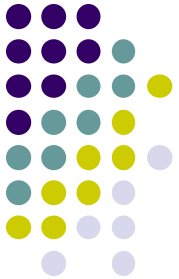
Analysis of Algorithms

Chapter 4.1, 4.2, 4.4, 4.5

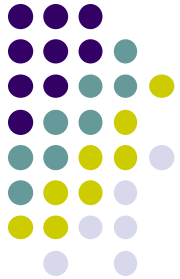


ROAD MAP



- **Decrease And Conquer**
  - Insertion Sort
  - Topological Sorting
  - Decrease By a Constant-Factor Algorithms
    - Binary Search
    - Fake-Coin Problem
    - Russian Peasant Multiplication
  - Variable-Size-Decrease Algorithms
    - Computing a Median and the Selection Problem
    - Interpolation Search

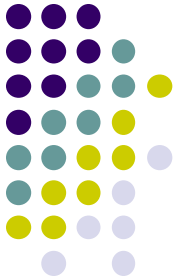
Decrease And Conquer



Decrease and conquer technique is based on

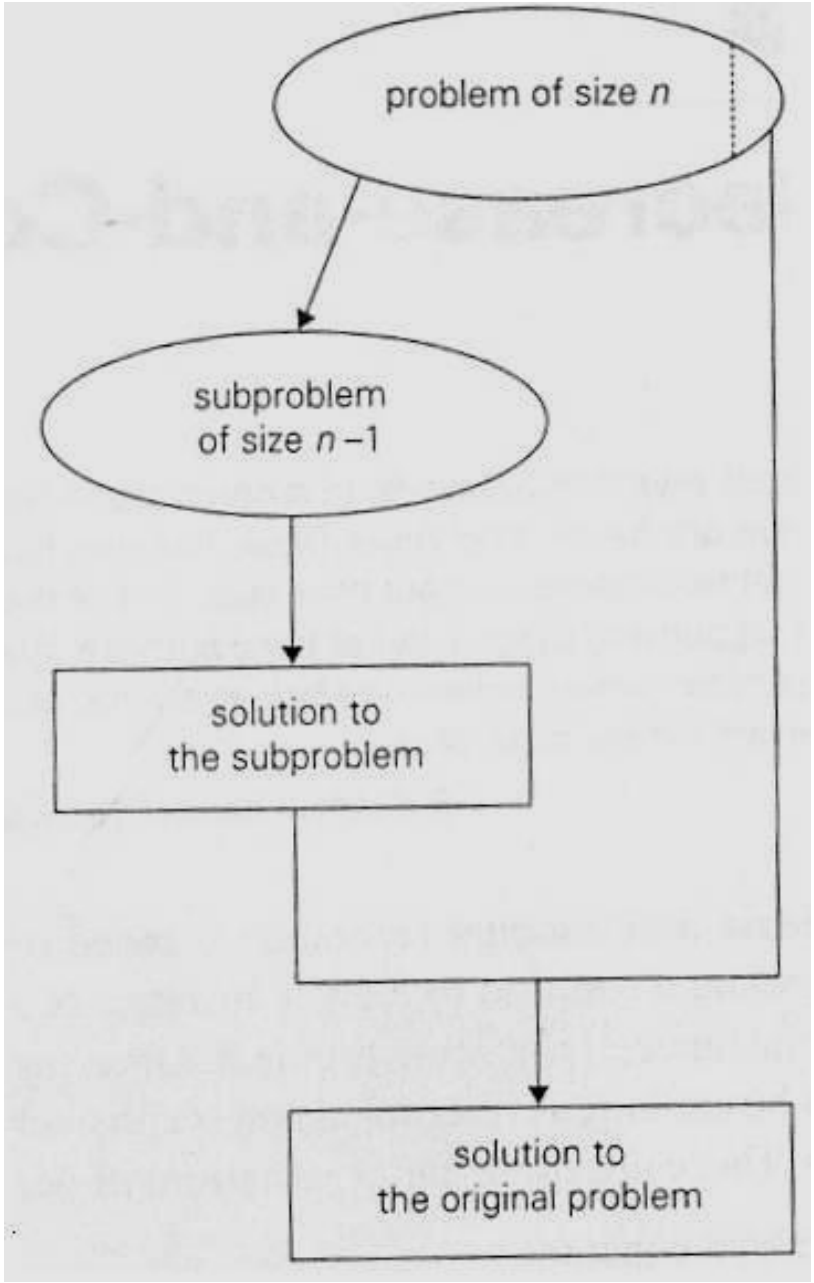
- To solve a *problem,* use a solution of a *smaller instance of the same problem*
- it can be done either top down (recursively) or bottom up (without a recursion)
- Variations of decrease and conquer :
  1. Decrease by a constant
  2. Decrease by a constant factor
  3. Variable size decrease

Decrease And Conquer



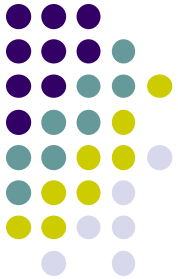
1. Decrease by a constant

- Size of an instance is reduced by the same constant on each iteration of the algorithm
  - typically this constant is equal to one



- **Decrease (by one) and conquer technique**

Decrease And Conquer



Example :Computing a^n for a positive integer a.

- Relationship between a solution to an instance of size *n* and size *n-1* is obtained by formula

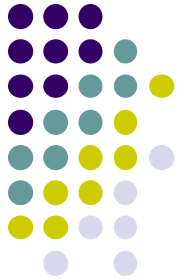
$$a^n = a^{n-1}.a \qquad\qquad f(n) = a^n$$

- can be computed topdown by using recursive definition

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

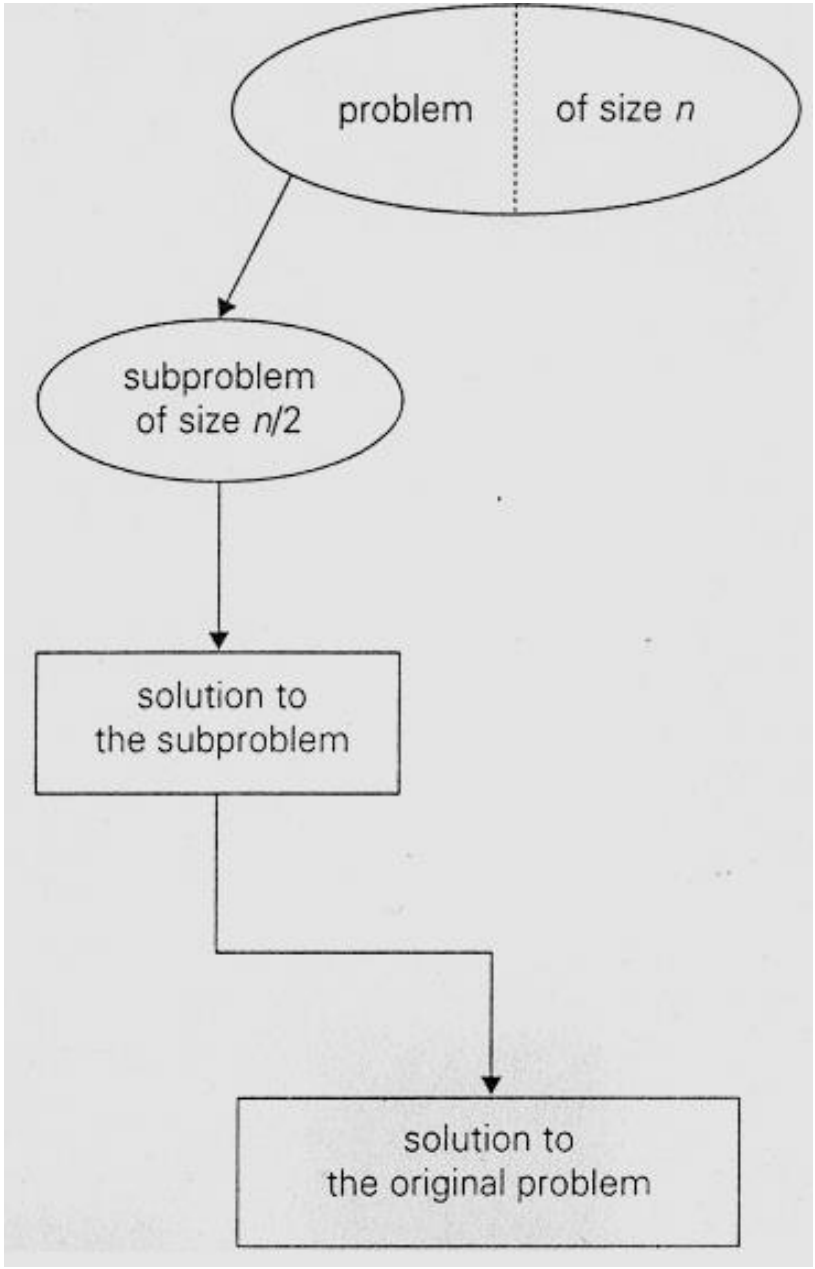
- can be computed bottom up by multiplying *a* by *n-1* times
  - same as brute force

Decrease And Conquer



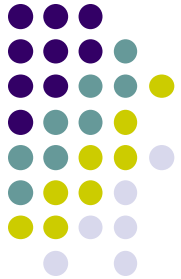
2. Decrease by a constant factor

- Reduce a problem's instance by the some constant factor on each iteration of the algorithm
  - in most applications this constant is two



- **Decrease (by half) and conquer technique**

Decrease And Conquer



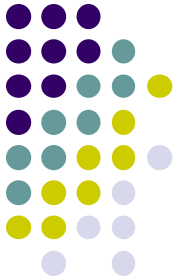
Example: Computing a^n for positive integer a

- If the instance of size *n* is to compute a^n, the instance of half its size will be to compute $a^{n/2}$

$$a^n = \left(a^{n/2}\right)^2$$

Does this work for all integers ?

Decrease And Conquer



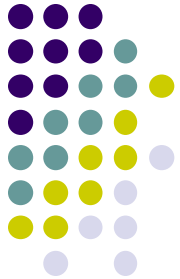
The formula is different for odd or even integers

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if n is even and positive} \\ \left(a^{(n-1)/2}\right)^2 . a & \text{if n is odd and greater than 1} \\ a & \text{if n = 1} \end{cases}$$

The runtime of the algorithm is *O(logn)*

Why??

Decrease And Conquer



3. Variable size decrease

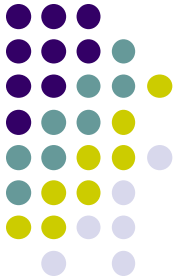
- A size reduction varies from one iteration of an algorithm to another
- EX: Euclid's algorithm for computing the greatest common divisor

```
gcd (m,n) = gcd (n, m mod n)
```

Arguments on right-hand side are always smaller than those on the left-hand side

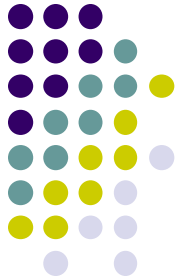
- **At least starting with the second iteration of the algorithm**

ROAD MAP



- **Decrease And Conquer**
  - **Insertion Sort**
  - Topological Sorting
  - Decrease By a Constant-Factor Algorithms
    - Binary Search
    - Fake-Coin Problem
    - Russian Peasant Multiplication
  - Variable-Size-Decrease Algorithms
    - Computing a Median and the Selection Problem
    - Interpolation Search

Sorting



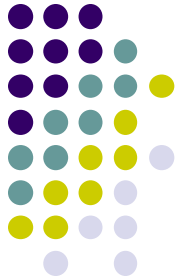
Definition :

Sorting an array *A[0 .. n-1]*

Which type of decrease-and-conquer is possible to use?

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decrease

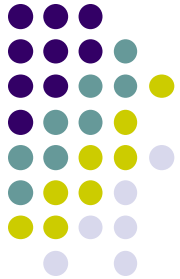
Sorting



Use decrease-by-a-constant:

- **Approach :**
  - The smaller problem: Sorting the array *A[0 .. n-2]*
    - *A[0] ≤ A[1] ≤ … ≤ A[n-2]*
  - After the smaller problem is solved
    - a sorted array of size *n-1 is obtained*
  - Need to find an appropriate position for *A[n-1]* among the sorted elements and insert it there
  - There are 3 alternative ways to do this …

Insertion Sort



1. Scan the sorted subarray from left to right until the first element greater than or equal to *A[n-1]* is found

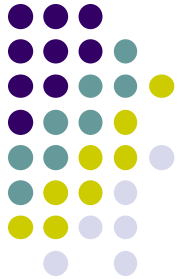
2. Scan the sorted subarray from right to left until the first element smaller than or equal to *A[n-1]* is found

Resulting algorithm is called s*traight insertion sort*

3. Use binary search to find an appropriate position for *A[n-1]* in the sorted portion of the array

Resulting algorithm is called *binary insertion sort*

Insertion Sort



- Can be implemented
  - top down, recursively
  - bottom up iteratively
    - More efficient
- Bottom up algorithm:
  - Loop: starting with *A[1]* and ending with *A[n-1]*
  - insert *A[i]* in its appropriate position
    - among the first *i* elements of the array that have been already sorted

Insertion Sort

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 of algorithm is comparison A[j] > v

Why not j≥0 ?

Insertion Sort

- Example :

89	**45**	68	90	29	34	17
45	89	**68**	90	29	34	17
45	68	89	**90**	29	34	17
45	68	89	90	**29**	34	17
29	45	68	89	90	**34**	17
29	34	45	68	89	90	**17**
17	29	34	45	68	89	90

Insertion Sort

- **Analysis :**
  - # of key comparison depends on nature of input
    - worst case

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

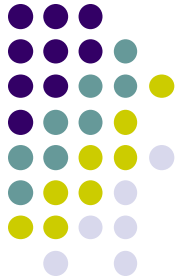
    - best case

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

    - average case

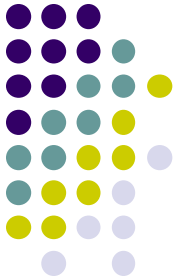
$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Insertion Sort



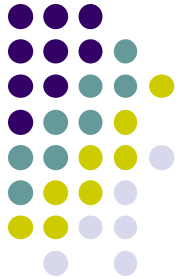
- **Discussion :**
  - In worst case insertion sort makes exactly the same number of comparisons as selection sort
  - For sorted arrays, insertion sort's performance is very good but we can not expect such convenient inputs
    - However, its performance is good on almost sorted arrays
      - Almost sorted arrays arise in a variety of applications
  - Its extension named ***shellsort*** gives a better algorithm for sorting large files

ROAD MAP



- **Decrease And Conquer**
  - Insertion Sort
  - **Topological Sorting**
  - Decrease By a Constant-Factor Algorithms
    - Binary Search
    - Fake-Coin Problem
    - Russian Peasant Multiplication
  - Variable-Size-Decrease Algorithms
    - Computing a Median and the Selection Problem
    - Interpolation Search

Topological Sorting



- **Definition :**
  - Given a directed graph, list vertices in an order
    - where for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends
  - No solution for a digraph which has a cycle
    - A graph must be a *dag (directed acyclic graph)*
  - Topological sorting may have several alternative solutions

Topological Sorting

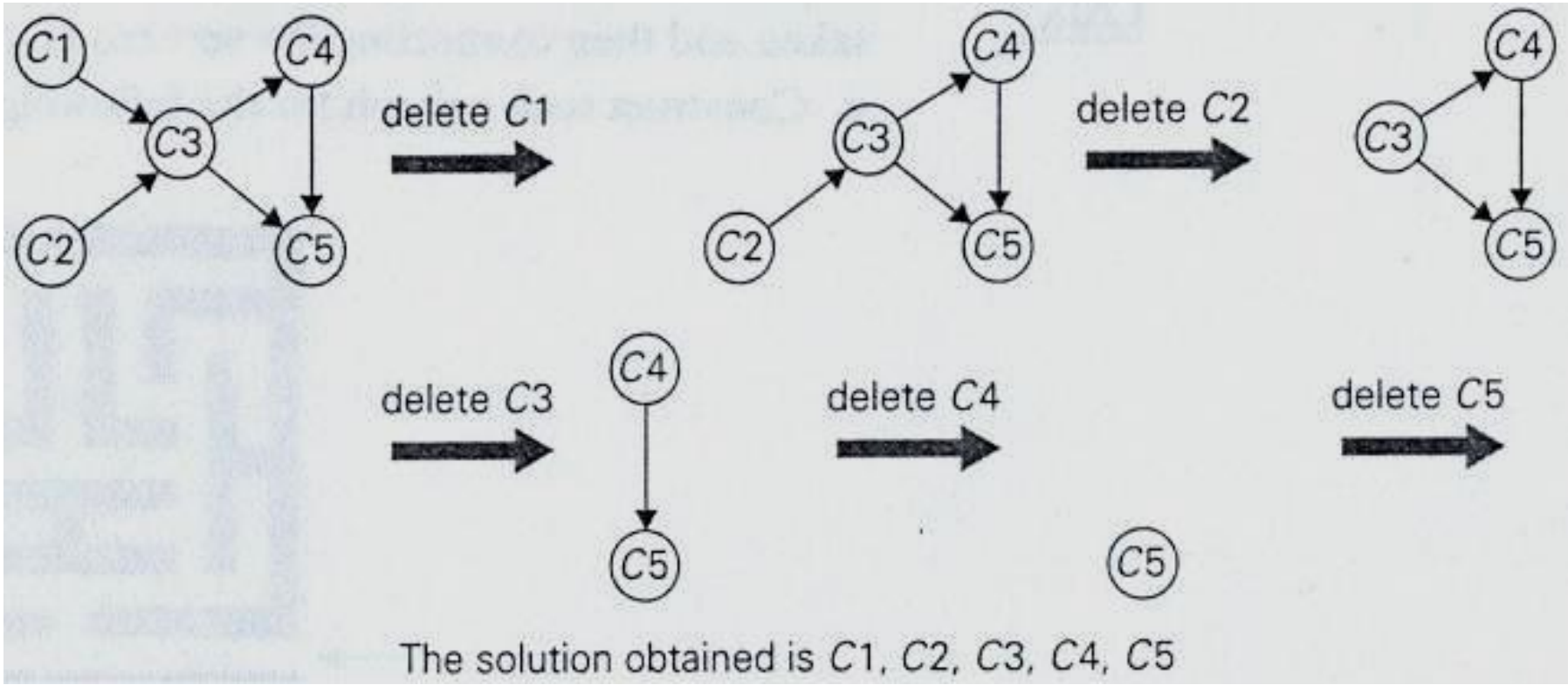
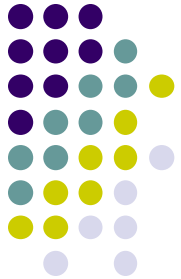


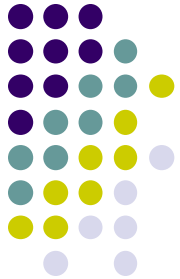
Illustration of the source removal algorithm for the topological sorting problem

ROAD MAP



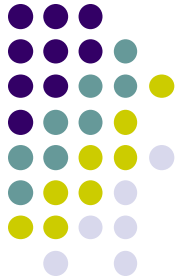
- **Decrease And Conquer**
  - Insertion Sort
  - Topological Sorting
  - **Decrease By a Constant-Factor Algorithms**
    - Binary Search
    - Fake-Coin Problem
    - Russian Peasant Multiplication
  - Variable-Size-Decrease Algorithms
    - Computing a Median and the Selection Problem
    - Interpolation Search

Decrease By a Constant-Factor Algorithms



- We will see the following algorithms based on decrease by a constant-factor idea
  - *Binary Search*
  - *Fake-Coin Problem*
  - *Russian Peasant Multiplication*
- These algorithms are fast
  - usually logarithmic
- A reduction by a factor other than two is rare

Binary Search



Binary search is a remarkably efficient algorithm for searching in a ***sorted*** array

An array A[0 .. n-1] and a search key *K* is given

Approach :

1. Comparing a search key *K* with the array's middle element A[m]

2. If they match, the algorithm stops

3. Otherwise same operation is repeated recursively for the first half of the array if K<A[m] and for the second half if K>A[m]

Binary Search

- *Example :*

Apply binary search to searching for K =70 in the array

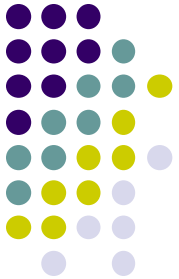
index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	*l*						*m*						*r*
iteration 2								*l*		*m*			*r*
iteration 3								*l,m*	*r*				

Binary Search

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
```

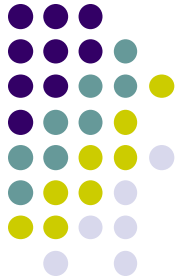
Binary Search



- **Analysis :**

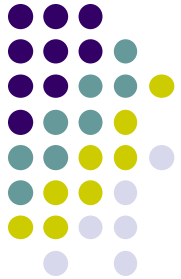
  What is the basic operation ?

Binary Search



- Basic operation is the ***comparison*** of the search key and an element of the array
- How many comparisons are made?
  - Depends on n
  - Also depends on the problem instance
- Requires best, worse and average case analysis

Binary Search



- Best case: When the search key is in the middle of the array.

$$C_b(n) = \Theta(1)$$

Binary Search

- Worst case: When the search key is not found in the array

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

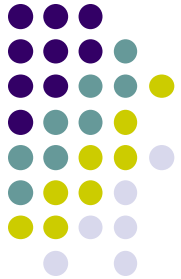
By assuming that n = 2^k and solving the recurrence by backward substitution, we get

$$C_w(2^k) = k + 1 = \log_2 n + 1.$$

In general

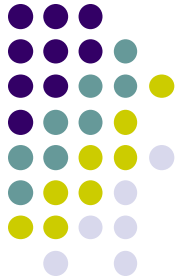
$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$

Binary Search



- What can you say about the average case ?

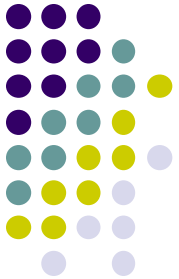
Binary Search



- An analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case

$$C_{avg}(n) \approx \log n$$

Binary Search

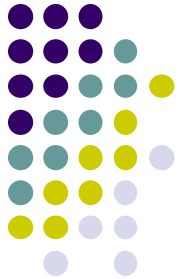


Discussion :

- advantages
  - be able to process large amounts of data
  - has a low cost per run

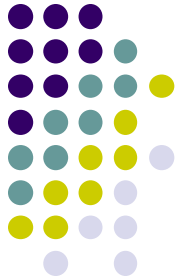
- disadvantages:
  - needs to be able to look at the whole array.
  - if there is too many data items then it requires a lot of memory.

Fake-Coin Problem



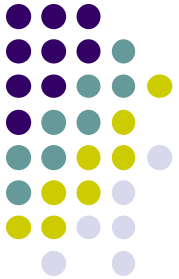
- There are several versions of fake-coin problem
- **Definition :**
  - Given a balance scale and n identically looking coins, one is ***fake***
    - Assume fake coin is lighter
    - In a balance scale, we can compare any two sets of coins
      - balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much
  - Find the fake coin

Fake-Coin Problem



- **Approach:**
  - Divide *n* coins into two piles of *n/2* coins each leaving one extra coin apart if n is odd and put the two piles on the scale
  - If the piles weigh the same, the coin put aside must be fake otherwise we can proceed in the same manner with the lighter pile which must be the one with the fake coin
  - After one weighing we are left to solve a single problem of half the original size so, this technique is divide by half and conquer rather than a divide and conquer algorithm

Fake-Coin Problem

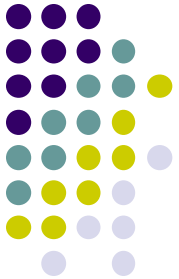


- Let *W(n)* be the number of weighings needed in worst case

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

- It is almost identical to the number of comparisons in binary search
  - The difference is the initial condition

Fake-Coin Problem

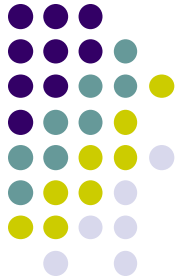


- Analysis :
  - Solution of the recurrence is

$$W(n) = \lfloor \log_2 n \rfloor$$

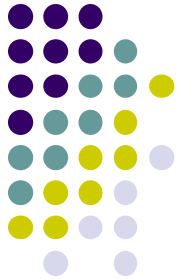
  - Is it possible to have more efficient algorithms?

Fake-Coin Problem



- **Discussion :**
  - We would be better of dividing the coins into three piles of about **`n/3`** coins each
  - After weighing two of the piles, we can reduce the instance size by a factor of *three*
  - We should expect the number of weighing to be about $\log_3 n$ which is smaller than $\log_2 n$
    - Can you tell by what factor?

Russian Peasant Multiplication



The problem: Compute the product of two positive integers

Can be solved by a decrease-by-half algorithm based on the following formulas.

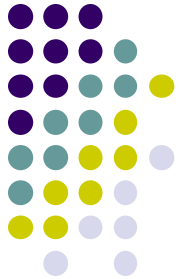
- If n is even

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

- If n is odd

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

Russian Peasant Multiplication

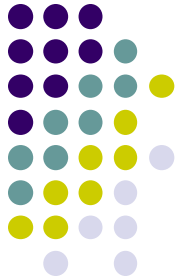


Compute 20 * 26

n	*m*	
20	26	
10	52	
5	104	
2	208	104
1	416	+
0	832	416
		520

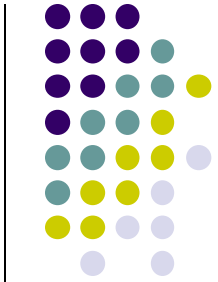
Note: Method reduces to adding m's values corresponding to odd n's.

ROAD MAP



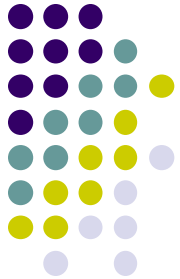
- **Decrease And Conquer**
  - Insertion Sort
  - Topological Sorting
  - Decrease By a Constant-Factor Algorithms
    - Binary Search
    - Fake-Coin Problem
    - Russian Peasant Multiplication
  - **Variable-Size-Decrease Algorithms**
    - Computing a Median and the Selection Problem
    - Interpolation Search

Variable-Size-Decrease Algorithms



- **Definition:**
  - Size reduction pattern varies from one iteration of the algorithm to another
  - Examples
    - Euclid's algorithm for computing gcd
    - Computing median
    - Selection problem

Selection Problem



Problem Definition :

- Find the k^{th} smallest element in a list of n numbers

  - This number is called the k^{th} order statistics
  - For $k=1$ or $k=n$ we can scan the list in question to find the smallest or largest element
  - A more interesting case is for $k=n/2$
    - This middle value is called ***median***
      - One of the most important quantities in mathematical statistics
  - The time of an algorithm that finds the kth smallest element by mergesort is $O(nlogn)$

Selection Problem

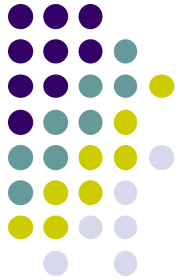
- **Approach :**

  To find the k^{th} smallest element we can divide elements into two subsets
  - Less than or equal to some value p
  - Greater than or equal to p

$$\underbrace{a_{i_1} \ldots a_{i_{s-1}}}_{\leq p} \quad \text{p} \quad \underbrace{a_{i_{s+1}} \ldots a_{i_n}}_{\geq p}$$

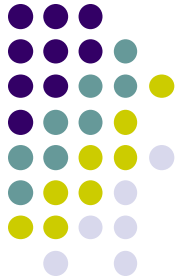
  - It is principal part of quicksort !
  - Can we take advantage of a list's partition ?

Selection Problem



- Let *s* be the partition's split position
  - Position of pivot *p*
- If **`s=k`**
  - the pivot *p* solves the selection problem
- If **`s>k`**
  - We look for k^{th} smallest element in the left part of the partitioned array
- If **`s<k`**
  - We can proceed by searching for the $(k\text{-}s)^{th}$ smallest element in its right part

Selection Problem

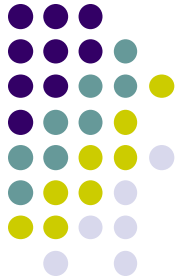


- Example :
  - Find median of the following list:

    4, 1, 10, 9, 7, 12, 8, 2, 15

    - $k = \lceil 9/2 \rceil = 5$
    - so we'll find the fifth smallest element in list
    - we assume that the elements of list are indexed from 1 to 9
    - we select the first element as pivot

Selection Problem



- Example :

4 1 10 9 7 12 8 2 15

2 1 4 9 7 12 8 10 15

since s = 3 < k = 5 proceed right part of list now k =2

9 7 12 8 10 15

8 7 9 12 10 15

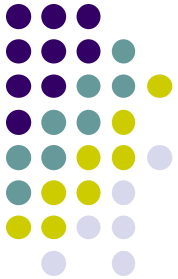
since s = 3 > k = 2 continue with left part of list k is still 2

8 7

7 8

now s = k = 2 and we stop – median is 8

Selection Problem

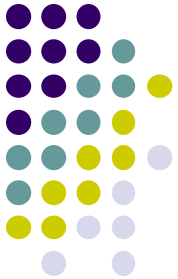


- **Analysis :**
  - We expect this algorithm to be more efficient than quicksort
    - It has to deal with a single subarray after a partition
      - Quicksort work on two of them
  - In the best case, splits always happen in middle of the array
  - So

$$C(n) = C(n/2) + (n+1)$$

$$C(n) \in \Theta(n)$$

Selection Problem



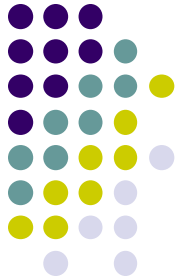
- **Analysis :**
  - In the worst case, the reduction of one element is achieved after each partition
  - So,

$$C(n) = C(n-1) + (n+1)$$

$$C(n) \in \Theta(n^2)$$

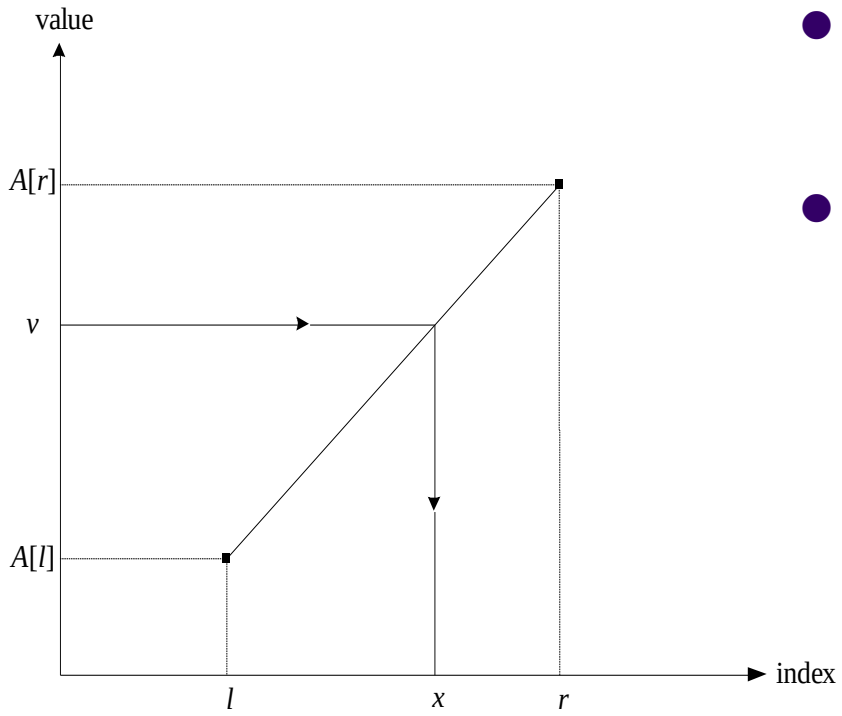
  - What about the average case?

Selection Problem



- **Discussion :**
  - An algorithm always works in linear time have been discovered
    - It is too complicated to be recommended for practical applications
  - Partitioning based algorithm solves more general problem
    - identifies the *k* smallest and *n-k* largest elements of a given list
    - not just the value of its k^{th} smallest element

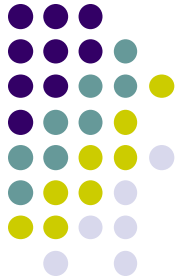
Interpolation Search



- Searches a sorted array similar to binary search
- Estimates location of the search key in A[l..r] by using its value x.
  - Specifically, the values of the array's elements are assumed to grow linearly from A[l] to A[r] and the location of x is estimated as:

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$

Interpolation Search



- Efficiency

  average case: $C(n) < \log_2 \log_2 n + 1$

  worst case: $C(n) = n$

- Preferable to binary search only for VERY large arrays and/or expensive comparisons