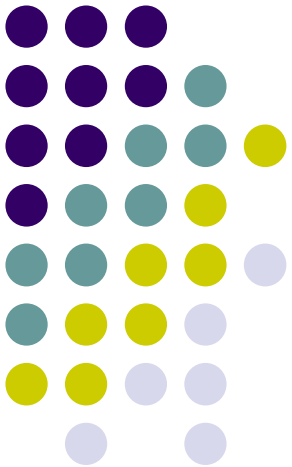
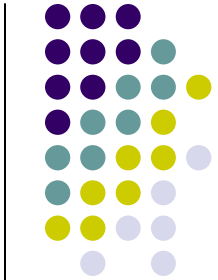
Introduction to Algorithm Design

Lecture Notes 6

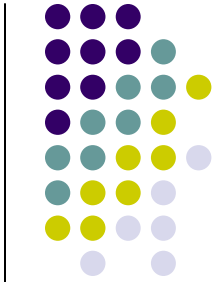


ROAD MAP



- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - Algorithms For Generating Combinatorial Objects
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

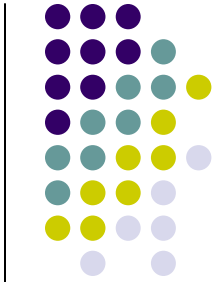
Decrease And Conquer



Decrease and conquer technique is based on

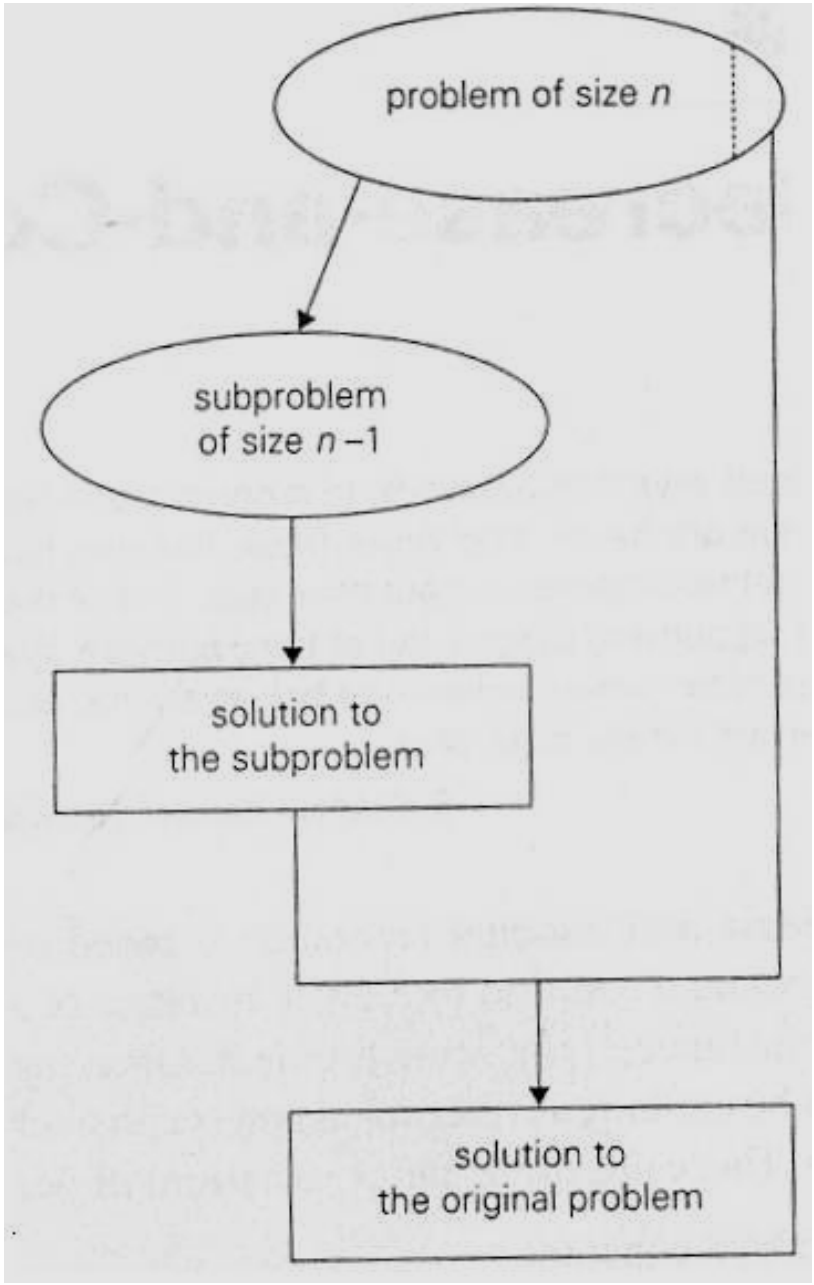
- To solve a *problem,* use a solution of a *smaller instance of the same problem*
- it can be done either top down (recursively) or bottom up (without a recursion)
- Variations of decrease and conquer :
  1. Decrease by a constant
  2. Decrease by a constant factor
  3. Variable size decrease

Decrease And Conquer



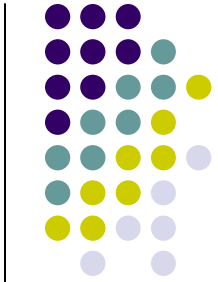
1. Decrease by a constant

- 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

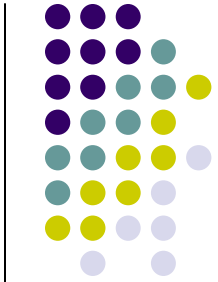
$$a^n = a^{n-1}.a \qquad\qquad f(n) = a^n$$

- can be computed topdown by using recursive definition

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

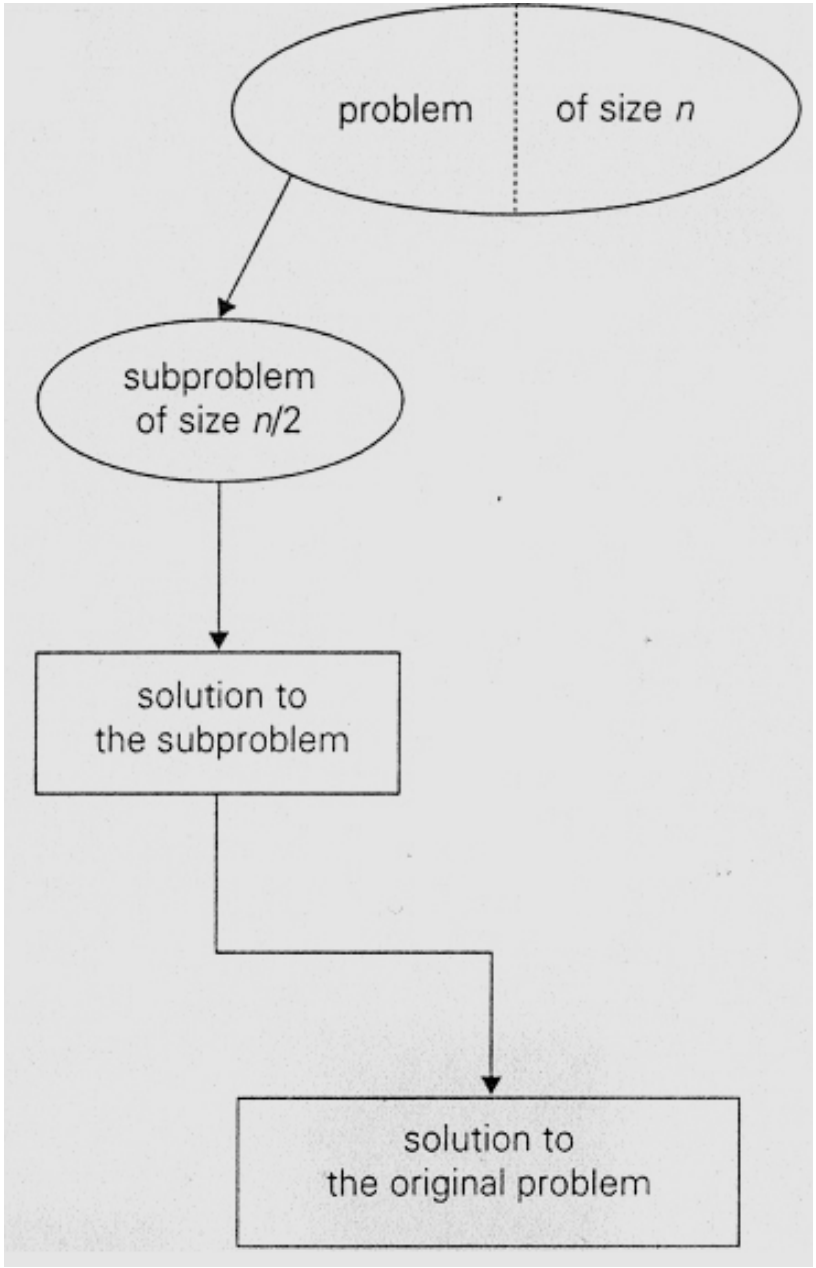
- can be computed bottom up by multiplying *a* by *n-1* times
  - same as brute force

Decrease And Conquer



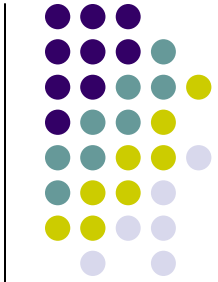
2. Decrease by a constant factor

- 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



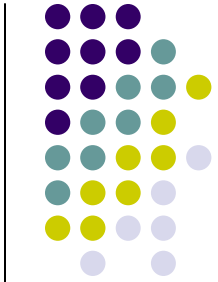
Example: Computing a^n for positive integer a

- 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

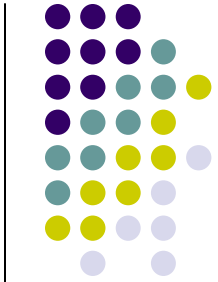
$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if n is even and positive} \\ \left(a^{(n-1)/2}\right)^2 .a & \text{if n is odd and greater than 1} \\ a & \text{if n = 1} \end{cases}$$

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

Why??

Is it the same with the algorithm based on divide & conquer idea ?

Decrease And Conquer

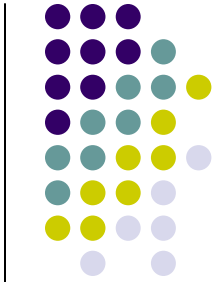


This algorithm is different from divide & conquer algorithm for solving two instances of exponentiation problem of size *n/2*

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

Is it efficient ? Why ?

Decrease And Conquer



3. Variable size decrease

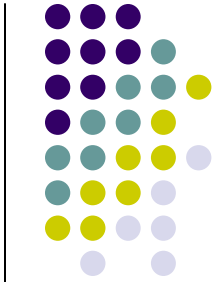
- A size reduction varies from one iteration of an algorithm to another
- EX: Euclid's algorithm for computing the greatest common divisor

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

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

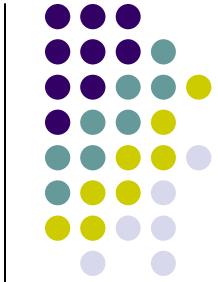
- **At least starting with the second iteration of the algorithm**

ROAD MAP



- **Decrease And Conquer**
  - **Insertion Sort**
  - **Depth-First Search**
  - **Breadth-First Search**
  - **Topological Sorting**
  - Algorithms For Generating Combinatorial Objects
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

Sorting



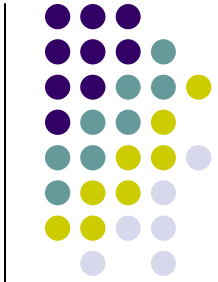
Definition :

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

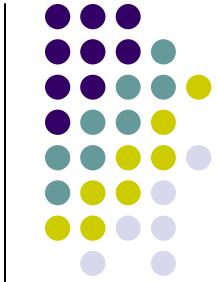
Sorting



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

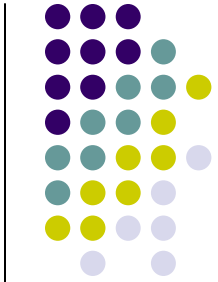
2. Scan the sorted subarray from right to left until the first element smaller than or equal to *A[n-1]* is found

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

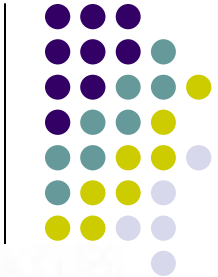
Resulting algorithm is called *binary insertion sort*

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



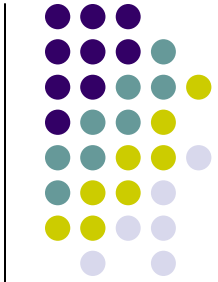
ALGORITHM *InsertionSort*$(A[0..n-1])$

```
//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 ?

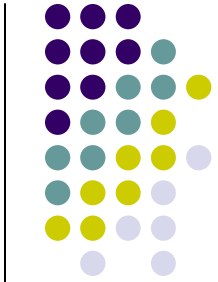
Insertion Sort



- 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

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

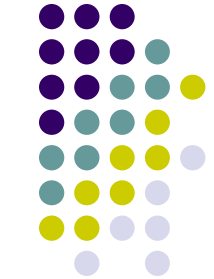
    - average case

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

Insertion Sort

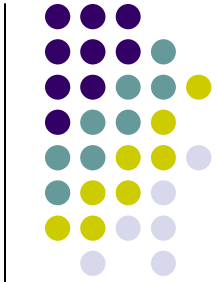
- **Discussion :**
  - In worst case insertion sort makes exactly the same number of comparisons as selection sort
  - For sorted arrays, insertion sort's performance is very good but we can not expect such convenient inputs
    - However, its performance is good on almost sorted arrays
      - Almost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

Depth-First Search (DFS)



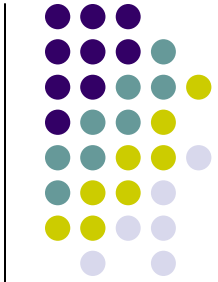
- It is one of the principal algorithms used to make traversals on graphs
- It is useful in investigating several important properties of a graph
  - Connectivity
  - Acyclicity
- Based on decrease-by-one technique

Depth-First Search



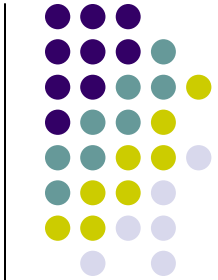
- **Approach :**
  - Start from an arbitrary vertex, mark it as visited
  - On each iteration, proceed to an unvisited vertex that is adjacent to the current one
    - which of the adjacent unvisited candidates is choosen?
  - This process continues until a dead end
    - a vertex with no adjacent unvisited vertices
  - At a dead-end, the algorithm back up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there
  - Algorithm halts after backing up to the starting vertex (being a dead-end)
    - Then all vertices in the same connected component as the starting vertex have been visited
  - If unvisited vertices still remain, DFS must be restarted at any one of them

Depth-First Search



- It is convenient to use a *stack* for DFS
  - We push a vertex on to the stack when the vertex is reached for the first time
  - We pop a vertex off the stack when it becomes a dead end
- The following is recursive algorithm

Depth-First Search

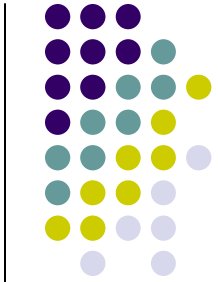


ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

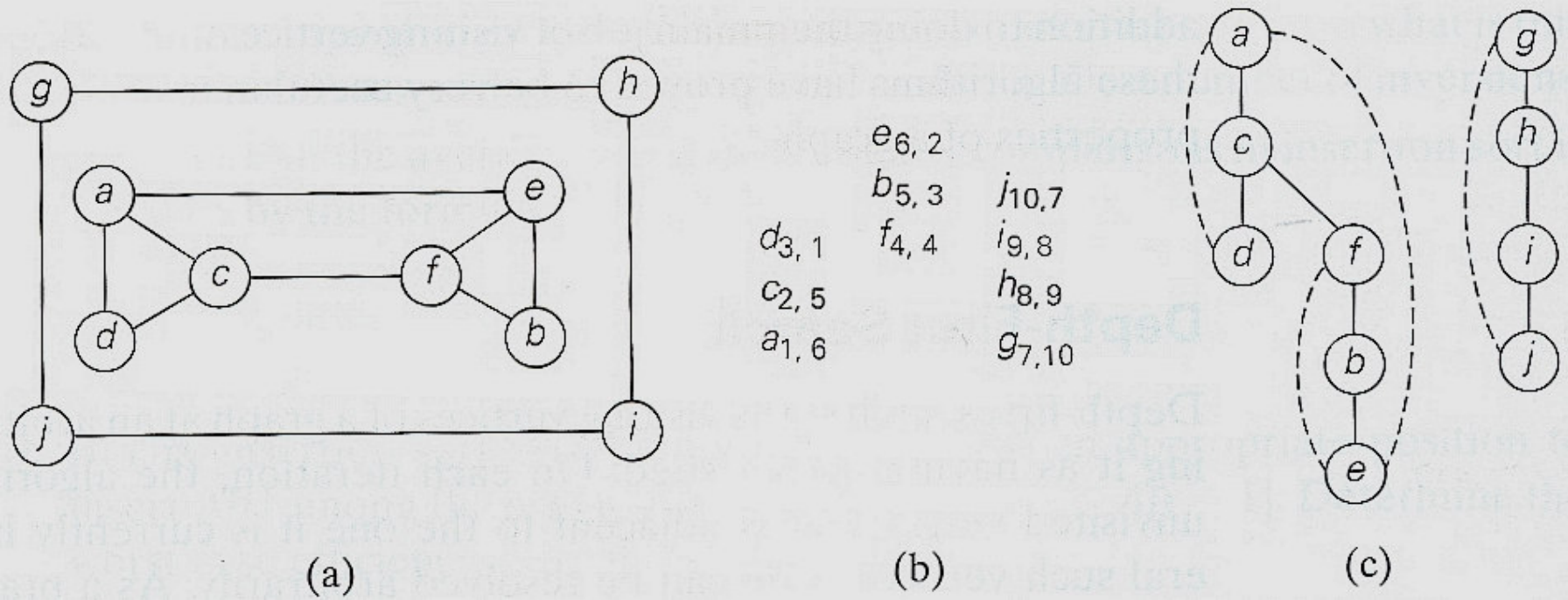
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v and
//assigns them the numbers in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)

Depth-First Search



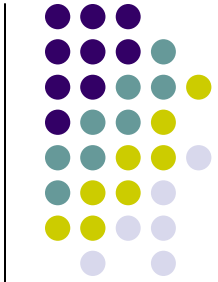
- Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached
  - called *tree edge*
- The edge leading to a previously visited vertex other than its immediate predecessor
  - called *back edge*

Depth-First Search (DFS)



a – Example of DFS traversal
b – Traversal's stack
c – DFS forest
tree edges shown with solid lines and back edges shown with dashed lines

Depth-First Search (DFS)



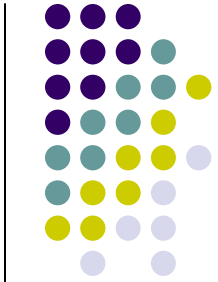
- **Analysis :**

  How efficient is DFS ?

  Algorithm takes time proportional to the size of the data structure used for representing the graph

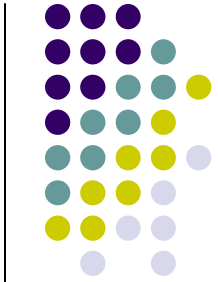
  - For adjacency matrix representation, time is in $\Theta(|V|^2)$
  - For adjacency list representation, time is in $\Theta(|V| + |E|)$

Depth-First Search (DFS)



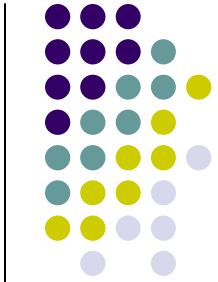
- **Discussion :**
  - DFS is efficient to check important properties of graphs
  - Elementary applications of DFS
    - Checking connectivity
    - Checking acyclicity

Breadth First Search (BFS)



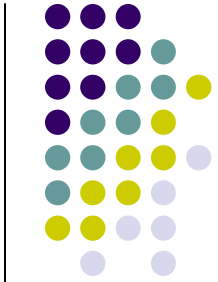
- **Definition :**
  - It is the other principal algorithm used to make traversals on graphs
  - Again based on decrease-by-one method
  - If DFS is a traversal for the brave, BFS is a traversal for the cautious

Breadth First Search

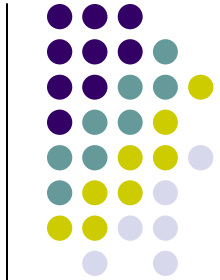


- **Approach :**
  - Visit all the vertices that are adjacent to a starting vertex
  - Then all unvisited vertices two edges apart from it and so on until all the vertices in the same connected component as the starting vertex are visited.
  - If there still remain unvisited vertices, algorithm has to be restarted at an arbitrary vertex of another connected component of the graph

Breadth First Search



- It is convenient to use a *queue*
  - different from DFS
- Queue is initialize with the traversal's starting vertex which is marked as visited.
- On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex
- Marks them as visited and adds them to the queue
- After that the front vertex is removed fro the queue

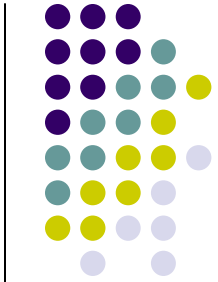


ALGORITHM *BFS*(G)

//Implements a breadth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        bfs(v)

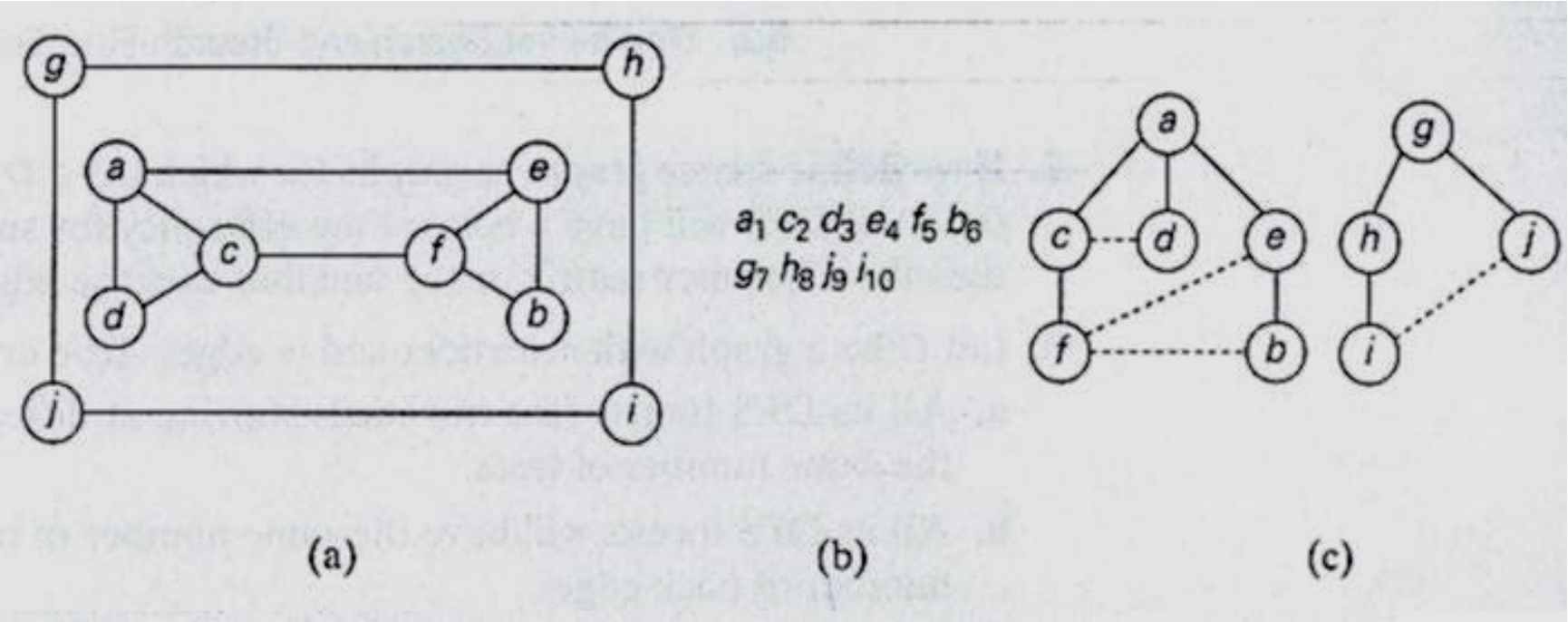
bfs(v)
//visits all the unvisited vertices connected to vertex v
//and assigns them the numbers in the order they are visited
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count* and initialize a queue with v
while the queue is not empty **do**
    for each vertex w in V adjacent to the front vertex **do**
        if w is marked with 0
            $count \leftarrow count + 1$; mark w with *count*
            add w to the queue
    remove the front vertex from the queue

Breadth First Search (BFS)



- Whenever a new unvisited vertex is reached for the first time, ithe vertex is attached as a child to the vertex is being reached from with an edge
  - called *tree edge*
- If an edge leading to a previously visited vertex other than its immediate predecessor is encountered, edge is
  - called *cross edge*

Breadth First Search

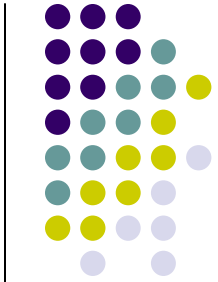


a – graph

b – traversal's queue

c – BFS forest

Breadth First Search

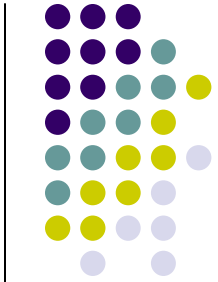


- **Discussion :**
  - BFS can be used to check connectivity and acyclicity of a graph as DFS
  - It can be helpful in some situations where DFS can not
    - Finding a path with fewest number of edges between two given vertices

Main Facts About DFS and BFS

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacent matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacent linked lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

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

Two possible algorithms:

- Use DFS
  - Reverse the order in which vertices become dead ends
  - If there is a back edge, the graph is not a dag
- The one we discussed in BIL222
  - Based on decrease-and-conquer technique
  - Find the vertex without incoming edge

Topological Sorting

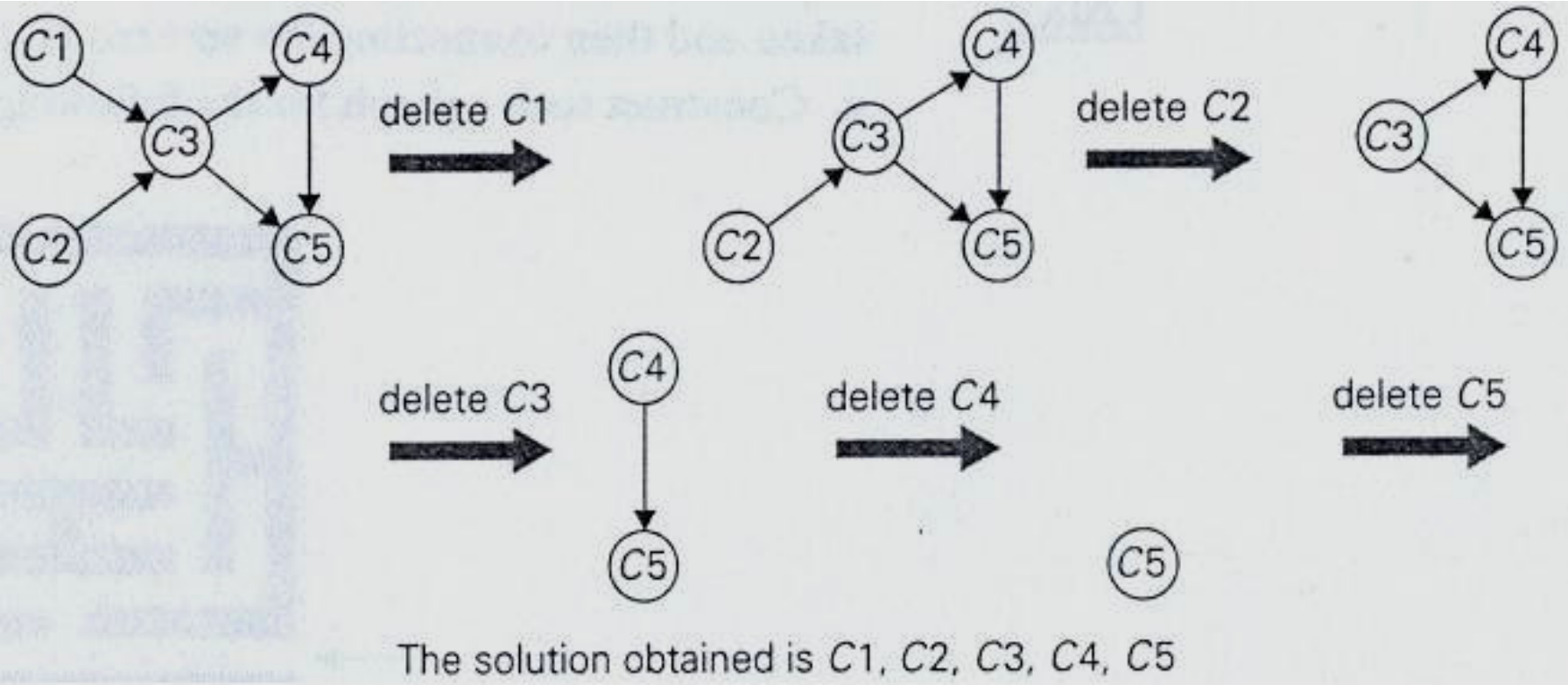
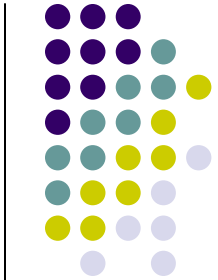


Illustration of the source removal algorithm for the topological sorting problem

ROAD MAP

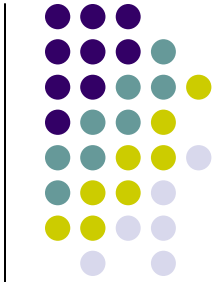


- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - **Algorithms For Generating Combinatorial Objects**
  - Decrease By a Constant-Factor Algorithms
  - Variable-Size-Decrease Algorithms

Algorithms for Generating Combinatorial Objects

- **Definition :**
  - Most important types of combinatorial objects
    - permutations
    - combinations
    - subsets of a given set
  - They typically arise in problems that require a consideration of different choices
    - Discussed before (TSP, Clique, Knapsack)
  - To solve these problems need to generate combinatorial objects
    - We are not interested in counting them

Generating Permutations



- Assume the set whose elements need to be permuted is the set of integers from *1 to n*
  - They can be interpreted as indices of elements in an *n-element* set $\{a_1, \ldots, a_n\}$
- What would the decrease-by-one technique suggest for the problem of generating all *n!* permutations?

Generating Permutations

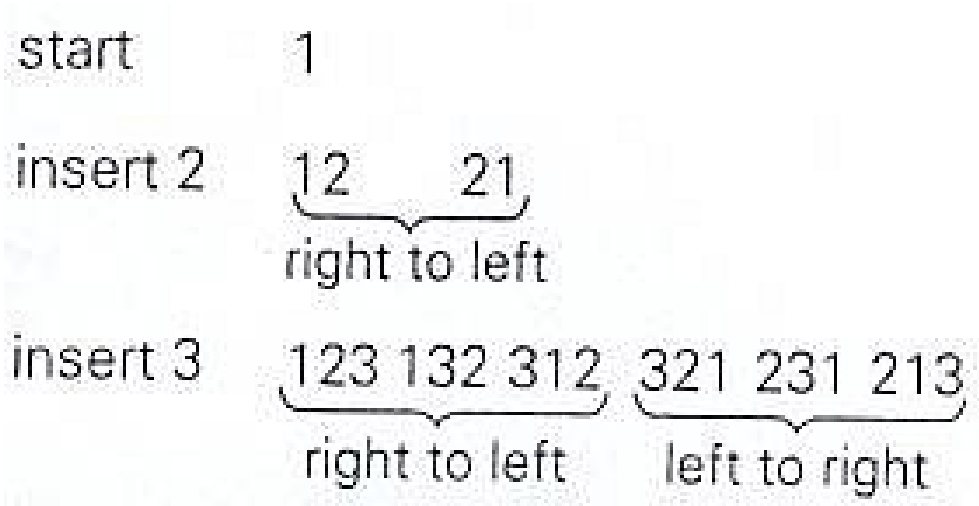
- **Approach :**
  - The *smaller-by-one* problem

    generating all *(n-1)!* permutations
  - Assuming that the smaller problem is solved
  - We can get a solution to the larger one
    - by inserting n in each of the *n* possible positions among elements of every permutation of *n-1* elements
    - There are two possible order of insertions
  - Total number of all permutations will be

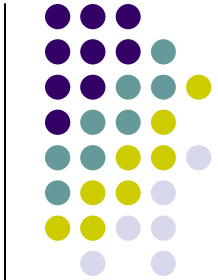
    n.(n-1) ! = n!

Generating Permutations

- We can insert *n* in the previously generated permutations
  - left to right
  - right to left



Generating Permutations



Approach:

- Assume all permutations of {*1,2,…,(n-1)} are available*
- Start with inserting n into *1,2,…,(n-1)* by moving right to left
- Then swicth direction every time a new permutation of {*1,2,…,(n-1)}* is processed

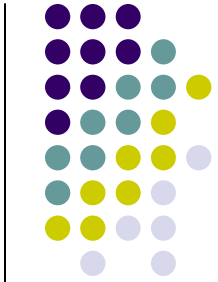
start	1	
insert 2	12	21
	right to left	
insert 3	123 132 312	321 231 213
	right to left	left to right

Generating Permutations

- This order satisfies ***minimal-change*** requirement
  - Each permutation is obtained from the previous one by exchanging only two elements
  - Benefical for algorithm's speed
- EX: Advantage of this order in TSP:
  - The length of the new tour can be calculated in constant time
    - By using the length of the previous tour

- This approach requires that all the permutations of *{1,2,…,(n-1)} are calculated already*
  - *Not easy to do!... (requires lots of space)*

Generating Permutations



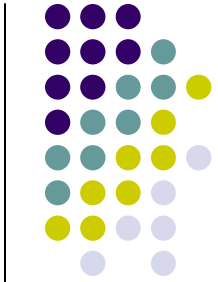
Another way to get the same ordering :

- Associate a direction with each component *k* in a permutation
- Indicate such a direction by a small arrow

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$$

- The component *k* is said to be mobile
  - if its arrow points to a smaller number adjacent to it
  - 3 and 4 are mobile
  - 2 and 1 are not
- The following algorithm uses this notion

Generating Permutations



ALGORITHM *Johnson Trotter (n)*

// Implements Johnson-Trotter algorithm for generating permutations

// Input : A positive integer *n*

// Output : A list of permutations of {1, … , n}

Initialize the first permitation with $\overleftarrow{1}\ \overleftarrow{2}\ \ldots\ \overleftarrow{n}$

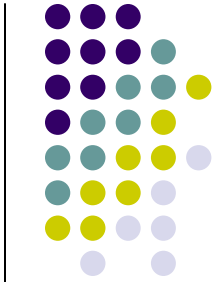
while there exists a mobile integer *k* do

  find the largest mobile integer *k*

  swap *k* and the adjacent integer its arrow points to

  reverse the direction of all integers that are larger than k

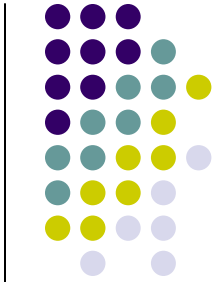
Generating Permutations



- An application of ***Johnson Trotter*** algorithm :

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{\mathbf{3}}$ $\quad$ $\overleftarrow{1}\ \overleftarrow{\mathbf{3}}\ \overleftarrow{2}$ $\quad$ $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{\mathbf{2}}$ $\quad$ $\overrightarrow{\mathbf{3}}\ \overleftarrow{2}\ \overleftarrow{1}$ $\quad$ $\overleftarrow{2}\ \overrightarrow{\mathbf{3}}\ \overleftarrow{1}$ $\quad$ $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

Generating Permutations

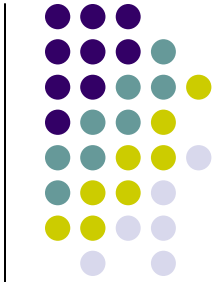


- **Analysis :**

  Runtime of the algorithm is $\Theta(n!)$

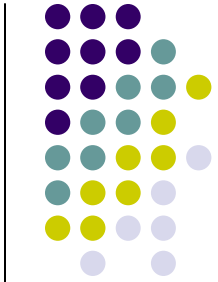
  Is it optimal?

Generating Permutations



- **Discussion :**
  - ***Johnson Trotter*** algorithm is one of the most efficient for generating permutations
  - In fact, because of the run time, it is horribly slow for all but very small values of *n*
  - However this is not the algorithm's *fault* but rather the fault of the problem
    - It simply asks to generate too many items

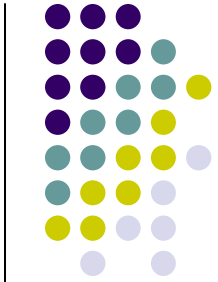
Clique problem



- Design an exhaustive-search for the *k-clique problem*
  - Check whether the graph has a clique of size *k*

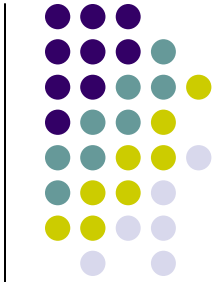
```
Generate all k-combinations of n vertices
    For each combination
        If combination is a complete graph
            Return true
    Return false
```

Generating Combinations



- Problem Definition :
  - Generate all r-combinations of a set of n elements
    - For some 0<r<n
- How can we solve the problem with decrease-by-one idea ?

Generating Combinations



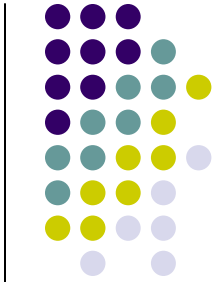
- Algorithm :

A is a vector of r elements

- Keeps an r-combination
- Filled from the last elemt to first

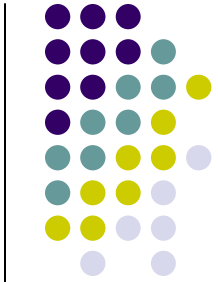
```
     procedure choose(b, c)
     comment choose c elements out of b..n
1.     if c = 0 then process(A)
2.       else for i := b to n − c + 1 do
3.         A[c] := i
4.         choose(i + 1, c − 1)
```

Generating Subsets



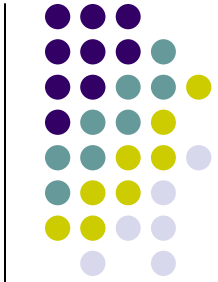
- Remember Knapsack Problem
  - Knapsack problem asks to find the most valuable subset of items that fits a knapsack of a given capacity
- Exhaustive search requires generating all subsets of a given set of items
- We will discuss generating all 2^n subsets of an abstract set $A = \{a_1, a_2, \ldots, a_n\}$
- How can we solve the problem with decrease-by-one idea ?

Generating Subsets



- All subsets of $A = \{a_1, a_2, \ldots, a_n\}$ can be divided into 2 groups
  - contain a_n and do not contain a_n
- Once we have a list of all subsets of $\{a_1, a_2, \ldots, a_{n-1}\}$
- We can *get all subsets of* $\{a_1, a_2, \ldots, a_n\}$ by adding to the list all its elements with a_n put into each of them
- The approach is not practical
  - Requires to calculate all subsets of $\{a_1, a_2, \ldots, a_{n-1}\}$
- There are other practical methods

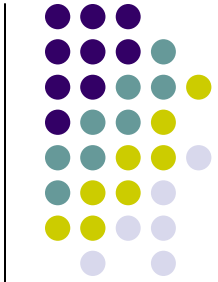
Generating Subsets



- Generating subsets bottom up

n	subsets
0	$\emptyset$
1	$\emptyset$ $\{a_1\}$
2	$\emptyset$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	$\emptyset$ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$ $\{a_1, a_2, a_3\}$

Generating Subsets

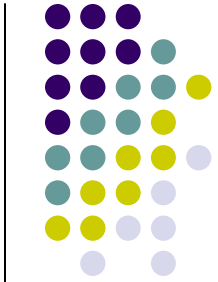


- Idea :
  - Assign to a subset the bit string in which
    - $b_i = 1$ if a_i belongs to the set
    - $b_i = 0$ if a_i does not belong to it
  - For set of a three-elements $\{a_1, a_2, a_3\}$

bit strings	000	001	010	011	100	101	110	111
subsets	$\emptyset$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

  - To generate subsets, generate all bit strings of length n
    - Generate binary numbers from 0 to 2^n-1

Generating Subsets



- Bit strings are generated are in the *lexicographic order* (in the two-symbol alphabet of 0 and 1)

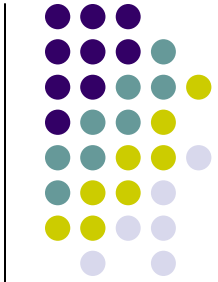
  000 001 010 011 100 101 110 111

- It is also possible to generate bit strings so that every one of them differs from its immediate prodecessor by only a single bit

  000 001 011 010 110 111 101 100

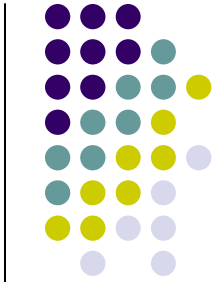
- Such a sequence of bit strings is called ***binary reflected Gray code***
- Gray code has many interesting properties and a few useful applications

ROAD MAP



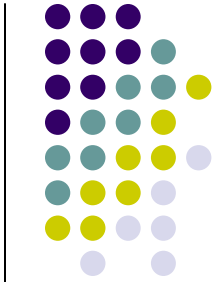
- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - Algorithms For Generating Combinatorial Objects
  - **Decrease By a Constant-Factor Algorithms**
  - Variable-Size-Decrease Algorithms

Decrease By a Constant-Factor Algorithms



- We have already seen several examples:
  - Binary search
  - Exponentiation by squaring
- We will see another algorithm based on decrease by a constant-factor idea
  - *Fake-Coin Problem*
- These algorithms are fast
  - usually logarithmic
- A reduction by a factor other than two is rare

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

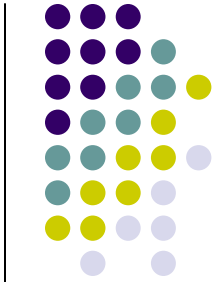
Fake-Coin Problem

- Analysis :
  - Solution of the recurrence is

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

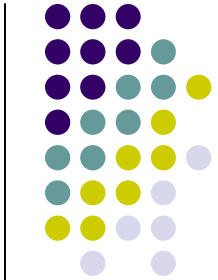
  - Is it possible to have more efficient algorithms?

Fake-Coin Problem



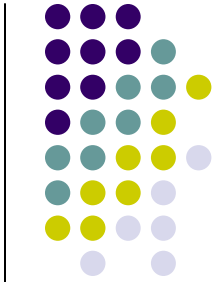
- **Discussion :**
  - We would be better of dividing the coins into three piles of about $n/3$ coins each
  - After weighing two of the piles, we can reduce the instance size by a factor of *three*
  - We should expect the number of weighing to be about $\log_3 n$ which is smaller than $\log_2 n$
 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

ROAD MAP



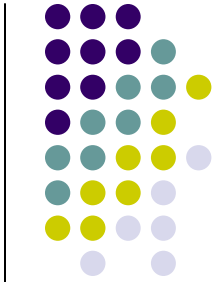
- **Decrease And Conquer**
  - Insertion Sort
  - Depth-First Search
  - Breadth-First Search
  - Topological Sorting
  - Algorithms For Generating Combinatorial Objects
  - Decrease By a Constant-Factor Algorithms
  - **Variable-Size-Decrease Algorithms**

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

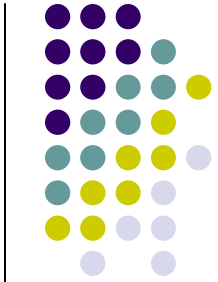
- **Approach :**

  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

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

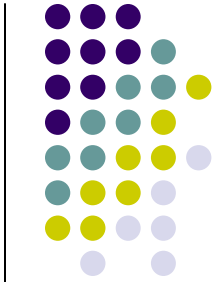
  - It is principal part of quicksort !
  - Can we take advantage of a list's partition ?

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

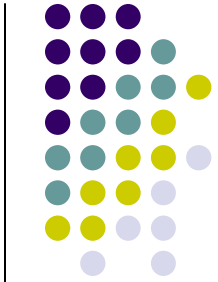


- Example :
  - Find median of the following list:

    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

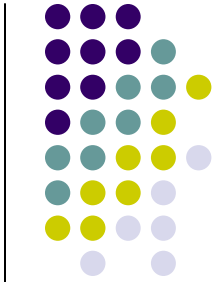
since s = 3 > k = 2 continue with left part of list k is still 2

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
  - Best splits: splits always happen in middle of the array
  - So

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

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

Selection Problem

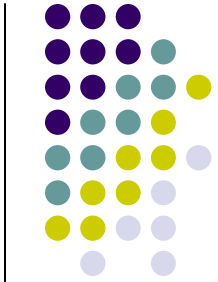
- **Analysis :**
  - In the worst case, the reduction of one element is achieved after each partition
  - So,

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

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

  - What about the average case?

Selection Problem



- **Discussion :**
  - Average case is linear
  - An algorithm always works in linear time have 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