Decrease-and-Conquer

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Chapter 5: Decrease-and-Conquer

Topics: What we will cover today

- Insertion sort
- Depth-first search and Breadth-first search
- Topological sorting
- Algorithms for generating combinatorial objects
- Decrease-by-a-constant-factor algorithms
- Variable-size-decrease algorithms

The Decrease-and-Conquer Design Strategy

- The decrease-and-conquer technique exploits the relationship between a solution to a problem instance and a solution to a smaller instance of the same problem.
- The relationship may be exploited top-down with recursion or bottom-up with iteration. We may refer to the latter as increase-and-conquer.
- The three major variations are:
  - decrease by a constant (often 1),
  - decrease by a constant factor (often half), and
  - variable size decrease.

Insertion Sort

- To sort the keys given in the array $A[0..n-1]$, we first sort the prefix $A[0..n-2]$, and then insert $A[n-1]$ into $A[0..n-2]$ at the right place.
- There are three ways to find the right place
  1. scan from $A[0]$ to $A[n-2]$ until the first $A[i]$ with $A[i] \geq A[n-1]$ or the prefix is exhausted;
  2. scan from $A[n-2]$ to $A[0]$ until the first $A[i] \leq A[n-1]$ or the prefix is exhausted;
  3. use binary search.

An Insertion Sort Algorithm

```
InsertionSort ( A[0..n − 1] )
  // Input: n keys in A[0..n − 1]
  for i ← 1 to n − 1 do
    // Insert A[i] to A[0..i − 1]
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
      A[j + 1] ← A[j]
      j ← j − 1
    od
    A[j + 1] ← v
  od
```

Insertion Sort: An 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

An Analysis of the Insertion Sort Algorithm

- Let $C(n)$ be the key comparison count.
- For the worst case (i.e., the input is already sorted reversely):

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

- For the best case (i.e., the input is already sorted):

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

An Analysis of the Insertion Sort Algorithm

- For the average case, it is known that

$$C(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

- Space efficiency: in-place ($O(1)$)
- Stability: yes
- Variation: binary insertion sort

High Level "Visualizing" of Sorting Algorithms

- To understand the differences between the different sorting algorithms we can divide the data array into a "prefix" and a "suffix".
- Then we can describe at a high level how a sorting algorithm does its work on the prefix and the suffix.

"Visualizing" Selection Sort

prefix $\leftarrow$ null;   suffix $\leftarrow A[0..n-1]$

while suffix $\neq$ null do
    swap the first element of suffix with
        a minimum element of suffix;
    make the first element of suffix to become
        the last element of prefix
od

"Visualizing" Bubble Sort

prefix $\leftarrow A[0..n-1]$;   suffix $\leftarrow$ null

while prefix $\neq$ null do
    sort every pair of adjacent prefix elements
        from left to right;
    make the last element of prefix to become
        the first element of suffix
od

"Visualizing" Insertion Sort

prefix $\leftarrow A[0]$; suffix $\leftarrow A[1..n-1]$

while suffix $\neq$ null do
  remove the first element of suffix and
    insert it into the prefix
od

Graph Algorithms

- Graph notations and terms:
  - Set of vertices V, number of vertices $|V|$.
  - Set of edges E, number of edges $|E|$.
  - Graph $G = (V, E)$.
  - Types of graphs: undirected, directed, and weighted.
  - Representations: adjacency matrix and adjacency lists.
  - Tree edge: direct edge between a parent and a child vertex.
  - Back edge in DFS: connects a vertex to an ancestor, other than the parent.
  - Cross edge in BFS: connects a vertex to a visited vertex other than its immediate predecessor.

Graph Traversal

- Many problems require processing of all graph vertices (and edges in a systematic fashion.
- Graph traversal algorithms:
  - There are two systematic ways to visit all the vertices of a graph: depth-first search (DFS) and breadth-first search (BFS).

Depth-First Search

- For each connected component, visit a given vertex. On the next search, visit an unvisited vertex adjacent to the vertex it has just visited; if there is no such vertex, back up to where it came from and re-start the search.
- Uses a stack:
  - A vertex is pushed onto the stack when it is reached for the first time.
  - A vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent, unvisited vertex.
- "Redraws" graph in a tree-like fashion (with tree edges and back edges for undirected graph).

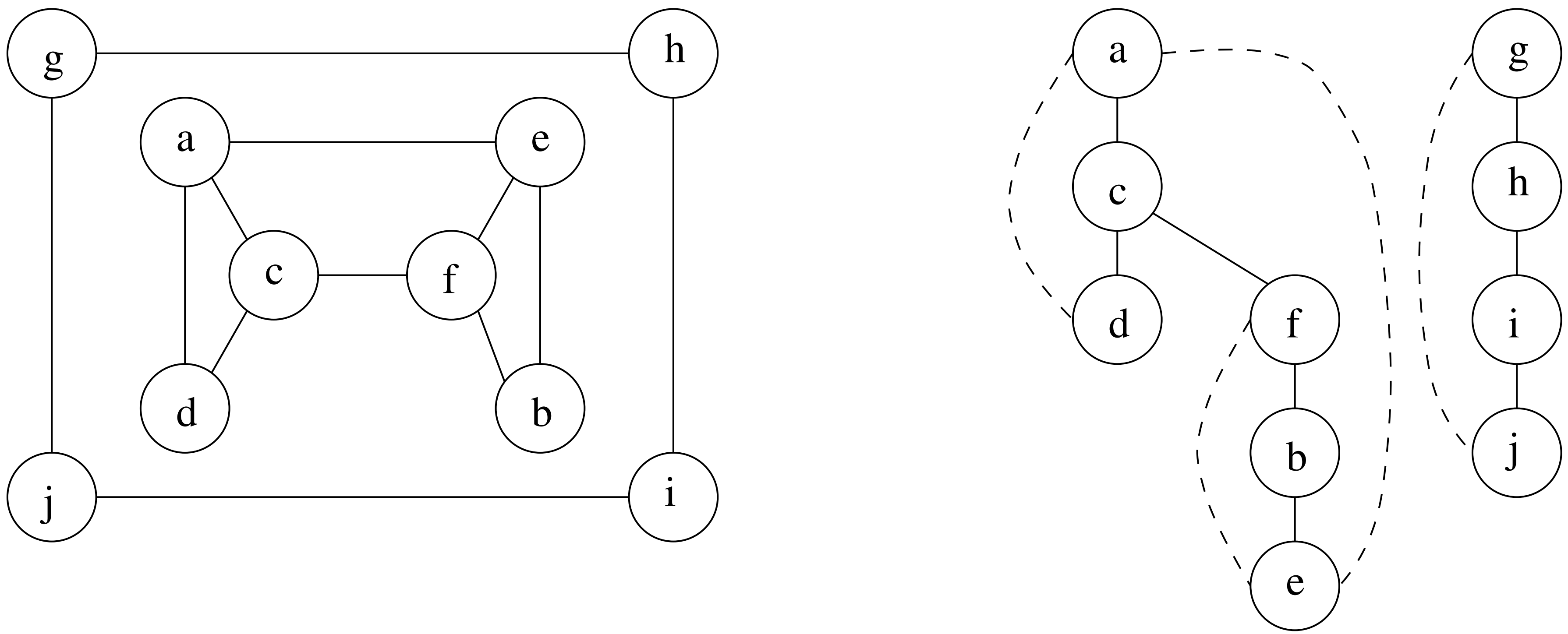
A Depth-First Search Algorithm

```
DFS( G )                    // G = (V, E), |V| = n
    for each vertex v in V mark v as unvisited        // Initialization
    for each vertex v in V
        if v is unvisited then dfs( v )
end
```

```
dfs( v )
    mark v as visited
    for each vertex w adjacent to v
        if w is unvisited then dfs( w )
end
```

A Depth-First Search Example

- Dashed lines: back edges.



Notes on Depth-First Search

- DFS can be implemented with graphs represented as adjacency matrices and adjacency lists.
- It yields two distinct orderings of vertices.
  - Order in which vertices are first encountered (pushed onto stack).
  - Order in which vertices become dead-ends (popped off stack).
- Applications
  - Checking connectivity, finding connected components
  - Checking acyclicity
  - Searching state-space of problems for solution (AI)

An Analysis of the Depth-First Search Algorithm

- The cost is clearly proportional to the size of the data structures used to represent the graph.
- For the adjacency matrix representation, every adjacency row has |V| entries. Thus the traversal time is in $\Theta(|V|^2)$.
- For the adjacency lists representation, the total number of adjacency lists entries is $2|E|$. Thus the traversal time is $\Theta(|V| + |E|)$.

Breadth First Search

- While depth first search tries to go as far as possible, breadth first search tries to stay as near as possible.
- It first visits the start vertex.
- It then visits all the vertices 1-edge away, then visits all the vertices 2-edge away, etc., until all the vertices in the component have been visited.
- While depth first search is greatly facilitated by the function call stack, breadth first is greatly facilitated by a queue.

A Breadth-First Search Algorithm (Initialization)

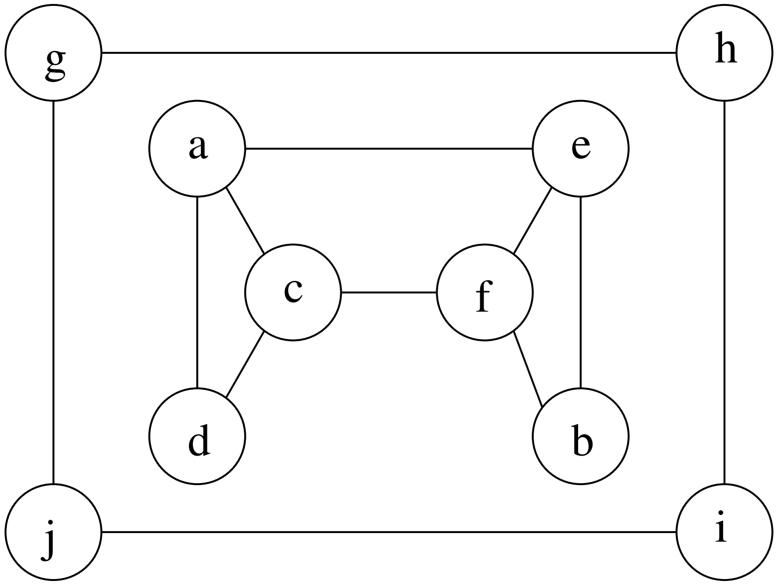
BFS(G) // $G = (V, E)$
    for each vertex v in V mark v as unvisited // Initialization
    for each vertex v in V
        if v is unvisited then *bfs*(v)
end

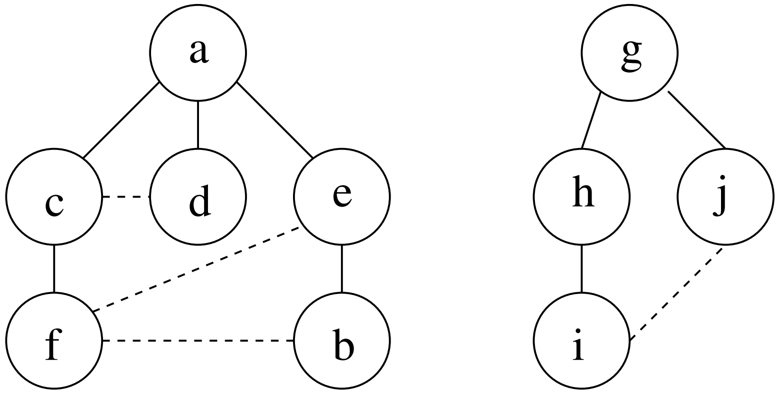
A Breadth-First Search Algorithm (Iteration)

```
bfs( v )
    mark v as visited; init_queue(); enqueue( v )
    while queue ≠ empty do
        u ← dequeue()
        for each vertex w adjacent to u do
            if w is unvisited then
                mark w as visited; enqueue( w )
            fi
        od
    od
end
```

A Breadth-First Search Example

- Dashed lines: cross edges.





An Analysis of the Breadth-First Search Algorithm

- Similar to depth first search, breadth-first search also has to check all the adjacent vertices of each vertex.
- The all cases time efficiency is also $\Theta(|V|^2)$ with adjacency matrices and $\Theta(|V| + |E|)$ with adjacency lists.

- It yields a single ordering of vertices (i.e., order added/deleted from queue is the same).
- Applications: same as DFS; but it can also find paths from a vertex to all other vertices with the smallest number of edges.

Topological Sorting

- A directed graph without cycles is a directed acyclic graph (dag).
- The vertices of a dag can be sorted topologically.
- A permutation of the n vertices $v_1, \cdots, v_n$ of the dag $G = (V, E)$

$$v_{i_1}, \cdots, v_{i_n}$$

  is a topological sort of the dag if (v_{i_p}, v_{i_q}) is not an edge whenever $p > q$.
- That is,

$$E \subseteq \{(v_{i_p}, v_{i_q}) : p < q\}.$$

Topological Sorting by Depth-First-Search

- Observe that when the function `dfs(v)` returns, all vertices that can be reached from v must have been visited.
- Thus the reverse order of the return of `dfs(v)` gives a topological sort.

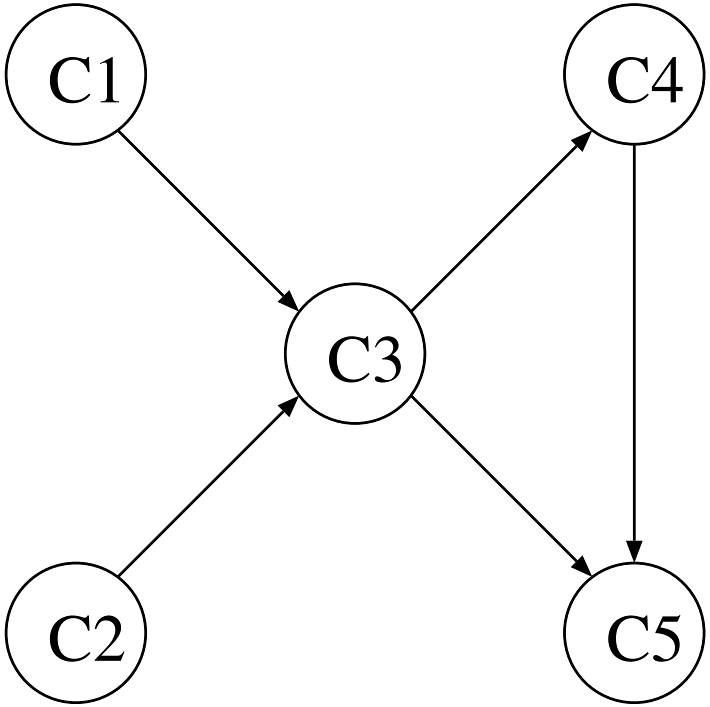
A Topological Sorting Example

- Example:

  A student needs to take 5 required courses: $\{C1, C2, C3, C4, C5\}$. The following pre-requisites need to be met: $C1$ and $C2$ have none; $C3$ requires $C1$ and $C2$; $C4$ requires $C3$; and $C5$ requires $C3$ and $C4$.

  The student can only take one course per semester.

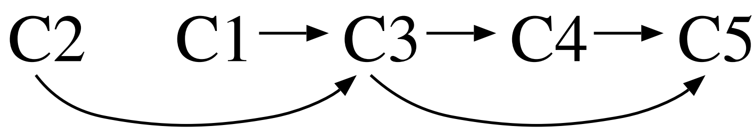
A Topological Sorting Example



The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:



A Topological Sort Algorithm by Depth-First-Search

```
// Produce a reverse topological sort
TS_DFS( G )                    // G = (V, E)
    for each vertex v in V mark v as unvisited
    for each vertex v in V
        if v is unvisited then dfs( v )
end

dfs( v )
    mark v as visited
    for each vertex w adjacent to v
        if w is unvisited then dfs( w )
    print v
end
```

Algorithms for Generating Combinatorial Objects

- Given a set of n objects, the most important types of combinatorial objects are
  1. the $n!$ permutations,
  2. the $\binom{n}{k}$ k-combinations,
  3. the 2^n subsets.
- Note that

$$\binom{n}{0} + \cdots + \binom{n}{n} = (1+1)^n = 2^n.$$

Generating Permutations

- Consider the problem of finding the $n!$ permutations of the sequence $1\ 2 \cdots n$.
- There are at least three methods: decrease by 1 (actually should be increase by 1), the Johnson-Trotter algorithm, and the lexicographic order method.
- The increase-by-1 algorithm enjoys the minimal-change property: each permutation can be obtained from its immediate predecessor by exchanging two adjacent digits.

An Illustration of the Increase-By-1 Algorithm: $3!$

- Generate 1.
- Insert 2 to 1 from right to left: 12, 21.
- Insert 3 to 12 from right to left: 123, 132, 312.
- Insert 3 to 21 from left to right: 321, 231, 213.
- Observe the minimal-change property of the permutations

$$123, 132, 312, 321, 231, 213.$$

An Illustration of the Increase-By-1 *Algorithm:* $4!$

- Insert 4 to 123 from right to left: 1234, 1243, 1423, 4123.
- Insert 4 to 132 from left to right: 4132, 1432, 1342, 1324.
- Insert 4 to 312 from right to left: 3124, 3142, 3412, 4312.
- Insert 4 to 321 from left to right: 4321, 3421, 3241, 3214.
- Insert 4 to 231 from right to left: 2314, 2341, 2431, 4231.
- Insert 4 to 213 from left to right: 4213, 2413, 2143, 2134.

Observe Minimal-Change Property of the Permutations

$1234, 1243, 1423, 4123, 4132, 1432, 1342, 1324, 3214, 3241, 3412, 4312,$

$4321, 3421, 3241, 3214, 2314, 2341, 2431, 4231, 4213, 2413, 2143, 2134.$

Mobile Digits of the Johnson-Trotter Algoritm

- Each element e in a permutation is given a left-to-right direction e. or a right-to-left direction .e.
- An element in a permutation is mobile if its direction points at a smaller element.
- For example, in the permutation $3..24..1$, the digits 3 and 4 are mobile, the digits 1 and 2 are not.

The Johnson-Trotter Permutation Generation Algorithm

```
JohnsonTrotter( n )                    // n is a positive integer
    initialize the first permutation with .1.2 . . . .n
    while previous permutation has a mobile element do
        find the largest mobile element k
        swap k with the element it is pointing to
        reverse direction of all elements larger than k
        add the new permutation to the list
    od
end
```

The Johnson-Trotter Algorithm: Illustration

.1.2.3.4 .1.2.4.3 .1.4.2.3 .4.1.2.3 4..1.3.2 .14..3.2 .1.34..2 .1.3.24.

.3.1.2.4 .3.1.4.2 .3.4.1.2 .4.3.1.2 4.3..2.1 3.4..2.1 3..24..1 3..2.14.

.23..1.4 .23..4.1 .2.43..1 .4.23..1 4..2.13. .24..13. .2.14.3. .2.13.4.

Lexicographic Ordering of Permutations

- Recall that we assume the n objects of permutation are numbers
$$1, \cdots, n.$$
- By treating a permutation as a base $n+1$ positional number, we can order the $n!$ permutations $12\cdots n, \cdots, n\cdots 21$ numerically, or lexicographically.

The Immediate Successor of a Permutation

- In lexicographic ordering, the immediate successor of the permutation $a_1 \cdots a_i \cdots a_n$ is $b_1 \cdots b_i \cdots b_n$ if and only if the following holds.
- $a_1 = b_1, \cdots, a_{i-1} = b_{i-1}$.
- The subscript i is the largest i such that $a_i < a_{i+1}$. That is,

$$a_i < a_{i+1} > a_{i+2} > \cdots > a_n.$$

- $b_i = \min\{a_k : k > i, a_k > a_i\}$.
- The sequence $b_{i+1} \cdots b_n$ is an increasing sort of the numbers $\{a_i, \cdots, a_n\} \setminus \{b_i\}$.

Lexicographically Ordered Permutations: Illustration

1234	1243	1324	1342	1423	1432	2134	2143
2314	2341	2413	2431	3124	3142	3214	3241
3412	3421	4123	4132	4213	4231	4312	4321

Generating Subsets

- There are at least three ways to generate the power set (the set of all subsets) of the set $\{1, 2, \cdots, n\}$: decrease-by-1, bit-vector, and binary reflected Gray node.

Generating Subsets: Decrease-By-One

- Let $P(n)$ be the power set of $\{1, \cdots, n\}$.
- The decrease-by-1 method is based on the observation that

$$P(n) = P(n-1) \cup \{S \cup \{n\} : S \in P(n-1)\}.$$

Generating Subsets: Bit Vectors

- The members of the set $S_n = \{1, \cdots, n\}$ may be represented by the bits of a n-bit vector: bit position i represents member $i + 1$.
- Thus each of the 2^n values $0..2^n - 1$, when written as n-bit base-2 numbers, gives a subset of S_n.
- Illustration:

0	1	2	3	4	5	6	7
000	001	010	011	100	101	110	111
$\{\}$	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{3\}$	$\{1,3\}$	$\{2,3\}$	$\{1,2,3\}$

Generating Subsets: Binary Reflected Gray Code

- The binary reflected Gray code generates bit vectors that enjoys the minimal change property.

Decrease-by-a-Constant Factor Algorithms

- Fake-coin identification problem
- Russian peasant method
- Josephus problem

Fake-Coin Identification Problem

- Among n identically looking coins, one is fake and is known to be lighter.
- Use an uncalibrated scale, identify the fake coin.

Fake-Coin Identification Problem

- One algorithm is to divide the n coins into at most three sets: $\lfloor n/2 \rfloor$ coins, $\lfloor n/2 \rfloor$ coins, and $\lceil n/2 \rceil - \lfloor n/2 \rfloor$ coin.
- If n is even, weigh the only two sets and the lighter set contains the fake coin.
- If n is odd, weigh the first two sets. If they weigh the same, the third set contains the fake coin; else the lighter set contains the fake coin.
- The problem size is now reduced by about half and the same procedure can be applied again.

An Analysis of the Fake-Coin Identification Algorithm

- The recurrence for the number of weighings $W(n)$ is

$$W(n) = W(\lfloor n/2 \rfloor) + 1, \;\; n > 1; \;\; W(1) = 0$$

  and the solution is $W(n) = \lfloor \log_2 n \rfloor$.
- It is known that a better algorithm with $W(n) \approx \log_3 n$ if the coins are divided into three sets of size about $n/3$.

The Russian Peasant Multiplication Method

- The product of two positive integers n and m can be found with halving, doubling, and adding:

$$nm = \begin{cases} \frac{n}{2} \times 2m, & n \text{ even;} \\ \frac{n-1}{2} \times 2m + m & n \text{ odd.} \end{cases}$$

- The operations of halving and doubling are simply 1-bit right-shift and 1-bit left-shift respectively.

The Russian Peasant Multiplication Method: Illustration

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

The Josephus Problem

- There are n persons numbered 1 to n standing in a circle.
- If n is even, eliminate all even-numbered persons;
- If n is odd, eliminate all even-numbered persons and the person numbered 1. The survivor who was numbered 3 becomes number 1.
- Repeat this process until a lone survivor is left.
- Who is the final survivor?

The Solution to the Josephus Problem

- Let $J(n)$ be the number of the final survivor.
- It can be verified that

$$J(2k) = 2J(k) - 1, \quad \text{and} \quad J(2k+1) = 2J(k) + 1,$$

  with the initial condition $J(1) = 1$.
- It turns out that

$$J(n) = \text{1-bit left cyclic shift of } n.$$

Variable-Size-Decrease Algorithms

- Computing a median and the selection problem
- Interpolation search
- Searching and insertion in a binary search tree
- The game of NIM

The Median and the Selection Problem

- The selection problem is about finding the k^{th} smallest element among n numbers.
- This number is called the k^{th} order statistic.
- When $k = \lceil n/2 \rceil$, the number is called the median of the given n numbers.
- For $k = 1$ or $k = n$, we simply scan the list to find the minimum or maximum value respectively.
- To sort the entire list to find the k^{th} order statistics is doing too much work.

Solving the Selection Problem by Partitioning

- An efficient algorithm (on average) is to use some partitioning algorithm like the one used in quicksort.
- Given an input array $A[1..n]$ (note that index starts at 1), let $A[s]$ be the pivot element.
- If $s = k$, $A[s]$ is the required answer.
- If $s > k$, then find the k^{th} smallest element among the $s - 1$ elements of the left partition $A[1..s - 1]$.
- If $s < k$, then find the $k - s^{th}$ smallest element among the $n - s$ elements of the right partition $A[s + 1..n]$.

An Analysis of the Selection by Partitioning Algorithm

- It is known that the average case efficiency is the same as if the size of array is reduced by about half in each iteration.
- If so the recurrence

$$C(n) = C(n/2) + \Theta(n)$$

  and the solution is $\Theta(n)$.
- The worst case efficiency is the same as quicksort: $\Theta(n^2)$.

Interpolation Search

- Recall that binary search looks for the key y among the partial array $A[l..r]$ by comparing y with the array element indexed by $\lfloor \frac{l+r}{2} \rfloor$.
- Interpolation search assumes the keys are uniformly distributed so it looks for the key y among the partial array $A[l..r]$ by comparing y with the array element indexed by

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

- The index $\lfloor x \rfloor$ is obtained by assuming that the points $(l, A[l]), (x, y), (r, A[r])$ lie on a (straight) line.

Interpolation Search

- If $y \neq A[x]$, the next iteration of the interpolation search narrows the search to $A[l..x-1]$ when $y < A[x]$ or to $A[x+1..r]$ when $A[x] < y$.
- Empirically interpolation search performs better than binary search when the range of search is large.

Searching and Insertion in a Binary Search Tree

- Recall that a binary search tree is either empty or the root key is larger than the keys in the left subtree but the smaller that the keys in the right subtree.
- The worst case time efficiency is $\Theta(n)$ when the tree is badly balanced but the average case time efficiency is $\Theta(\log n)$ as on the average the tree is reasonably balanced.

Searching and Insertion in a Binary Search Tree

- The search is done recursively:

```
BS( key, T )                    // looking for key in T
    if T = empty then return −1 fi
    if key = T.root then return T
    else
        if key < T.root then return BS(key, T.left)
        else
            return BS(key, T.right)
        fi
    fi
end
```

The Game of Nim: Rules

- Consider the game Nim when two players take turn to remove chips from 2 or more piles of chips.
- Each time a player must remove at least one chip but at most all the chips of a pile.
- The player who removes the last pile of chips wins the game.

The Game of Nim: A Fated Game

- It turns out that the winner is completely determined by the opening chip configurations and who moves first.
- To see this, assume there are $k > 1$ piles of chips and the number of chips are $n_1, \cdots, n_k$ with the following binary representations:

$$
\begin{array}{ccccc}
n_1 & = & b_{1,l} & \cdots & b_{1,0} \\
\vdots & = & \vdots & \ddots & \vdots \\
n_k & = & b_{k,l} & \cdots & b_{k,0} \\
\hline
\bigoplus & = & b_l & \cdots & b_0 \\
\hline\hline
\end{array}
$$

where "$\oplus$" denotes the bit-wise exclusively-or operation.

The Game of Nim: A Fated Game

- If all the bits $b_l, \cdots, b_0$ are zero, the player who moves next will lose; otherwise, the player who moves next will win.
- When some of the bits $b_l, \cdots, b_0$ are not zero, from which pile and how many chips should the player remove?
- Let the left-most 1-bit be b_i.
- There must be at least a $b_{j,i} = 1$ for some pile with n_j chips.
- For each of the bits $b_l, \cdots, b_0$ that is 1, flip the corresponding bits in n_j to obtain the number n'_j.
- Clearly, $n'_j < n_j$.
- The player should remove $n_j - n'_j$ chips from pile j.

Take Away Message on Decrease-and-Conquer

- The decrease-and-conquer technique exploits the relationship between a solution to a problem instance and a solution to a smaller instance of the same problem.
- The relationship may be exploited top-down with recursion or bottom-up with iteration. We may refer to the latter as increase-and-conquer.
- The three major variations are:
  - decrease by a constant (often 1),
  - decrease by a constant factor (often half), and
  - variable size decrease.