CSC 8301- Design and Analysis of Algorithms

Lecture 5

Decrease and Conquer
Algorithm Design Technique

Decrease-and-Conquer

This algorithm design technique is based on exploiting a relationship between a solution to a given instance of the problem in question and its smaller instance.

Once such a relationship is found, it can be exploited either top down (usually but not necessarily recursively) or bottom up.

It's probably the first alternative to brute force one should try in facing an unfamiliar problem.

3 Types of Decrease-and-Conquer

- *Decrease by a constant* (usually by 1):
  - insertion sort
  - algorithms for generating permutations, subsets
- *Decrease by a constant factor* (usually by half)
  - binary search
  - exponentiation by squaring
- *Variable-size decrease*
  - Euclid's algorithm $gcd(m,n) =$
  - selection by partition

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Decrease by one:

  $a^n =$

- Decrease by half:

  $a^n =$

Insertion Sort

To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]

Example: Sort 6, 4, 1, 8, 5

Usually implemented bottom up (nonrecursively)

6 | <u>4</u> 1 8 5

Pseudocode of 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
```

Analysis of Insertion Sort

```
for $i \leftarrow 1$ to $n - 1$ do
    $v \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ and $A[j] > v$ do
        $A[j + 1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j + 1] \leftarrow v$
```

- Time efficiency

  $C_{worst}(n) =$

  $C_{best}(n) =$

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

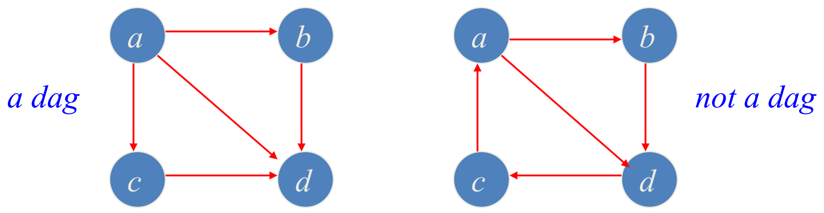
Analysis of Insertion Sort

```
for $i \leftarrow 1$ to $n - 1$ do
    $v \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ and $A[j] > v$ do
        $A[j + 1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j + 1] \leftarrow v$
```

- Space efficiency: in-place
- Stability: yes
- Best elementary sorting algorithm overall

Dags and Topological Sorting

DAG: Directed Acyclic Graph (no directed cycles)



- Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)
- *Topological sorting:*
  - Linear ordering of vertices such that, for each edge, the start vertex appears before the end vertex
  - Possible on DAGs only

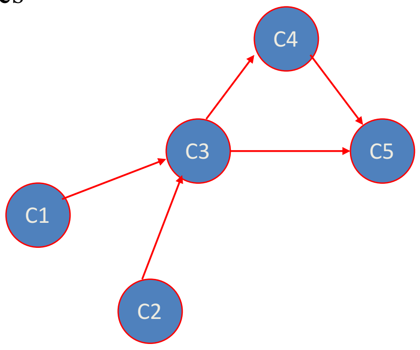
Topological Sorting Example

C1 and C2 have no prerequisites

C3 requires C1 and C2

C4 requires C3

C5 requires C3 and C4



Students can take only one course per term.

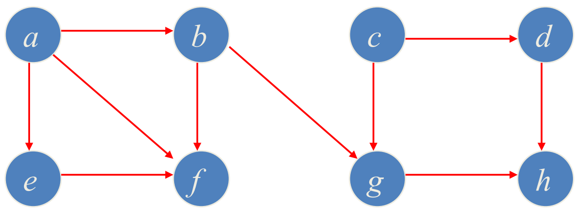
In what order should the students take the courses?

Topological Sorting: DFS-based Algorithm

DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered?→ NOT a dag!

Example:



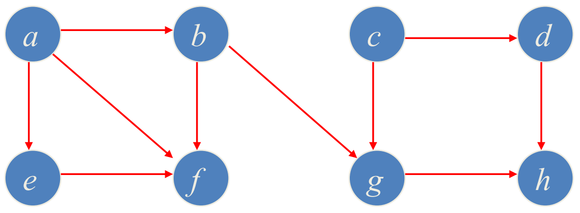
Efficiency:

Topological Sorting: Source Removal Algorithm

Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm

Generating Permutations

Decrease by one:

- Generate all (n-1)! permutations of {1, 2, …, n-1}
- Insert n in each of the n positions of each permutation

Example: $n = 3$

Generating minimal-change permutations

Minimal-change: each permutation differs from its predecessor by two adjacent elements

If $n = 1$ return 1; otherwise, generate recursively the list of all permutations of 12…n-1 and then insert n into each of those permutations by starting with inserting n into 12...n-1 by moving right to left and then switching direction for each new permutation

Example: $n = 3$

Other permutation generating algorithms

- Johnson-Trotter (p. 145 in the 3rd ed. of the textbook)
- Lexicographic-order algorithm (p. 146 in the 3rd ed.)
- Heap's algorithm (Problem 4 in Exercises 4.3 in the 3rd ed.)

Generating Subsets

Decrease by one:

- Generate all subsets of $\{1, 2, \ldots, n\text{-}1\}$, including the empty set
- Add n to each subset

Example: $n = 3$

Subset	Corresponding Bit String

Generating minimal-change subsets

Binary reflected Gray code: *minimal-change* algorithm for generating 2^n bit strings corresponding to all the subsets of an n-element set where $n > 0$

Minimal-change: each bit string differs from its predecessor by a single bit

```
If $n = 1$ make list $L$ of two bit strings 0 and 1
else
     generate recursively list $L1$ of bit strings of length $n-1$
     copy list $L1$ in reverse order to get list $L2$
     add 0 in front of each bit string in list $L1$
     add 1 in front of each bit string in list $L2$
     append $L2$ to $L1$ to get $L$
return $L$
```

Binary reflected Grey code example

Decrease-by-Constant-Factor Algorithms

In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)

Examples:

- Exponentiation by squaring
- Binary search and the method of bisection (pp. 460−463)
- Multiplication à la russe (Russian peasant method)
- Fake-coin puzzle

Binary Search

Very efficient algorithm for searching in sorted array:

$$\begin{array}{c} K \\ \text{vs} \\ A[0]\ \ .\ .\ .\ \ A[m]\ \ .\ .\ .\ \ A[n\text{-}1] \end{array}$$

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

Binary Search (non-recursive)

```
$l \leftarrow 0$;   $r \leftarrow n-1$
while $l \le r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if  $K = A[m]$  return $m$
    else if $K < A[m]$  $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1
```

Worst case recurrence relation:

Notes on Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$
- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

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:

For even values of *n*:

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

For odd values of *n*:

Example of Russian Peasant Multiplication

Compute 20 * 26

n	*m*
20	26

Fake-Coin Puzzle (simpler version)

There are *n* identically looking coins one of which is fake.
There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much). Design an efficient algorithm for detecting the fake coin. Assume that the fake coin is known to be lighter than the genuine ones.

Decrease by factor 2 algorithm

Variable-Size-Decrease Algorithms

In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another

Examples:

- Euclid's algorithm for greatest common divisor
- Partition-based algorithm for selection problem
- Some algorithms on binary search trees

Euclid's Algorithm

Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Ex.: gcd(80,44) = gcd(44,36) = gcd(36, 12) = gcd(12,0) = 12

One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations.

Hence, $T(n) \in O(\log n)$

Selection Problem

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

- $k = 1$ or $k = n$

- *median*: $k = \lceil n/2 \rceil$
  Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = ?

The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

Algorithms for the Selection Problem

The sorting-based algorithm: Sort and return the *k*-th element
Efficiency (if sorted by mergesort): Θ(*n*log *n*)

A faster algorithm is based on the array *partitioning*:

	s	
all are ≤ A[*s*]		all are ≥ A[*s*]

Assuming that the array is indexed from 0 to *n*-1 and *s* is a split position obtained by the array partitioning:

If $s = k$-1, the problem is solved;

if $s > k$-1, look for the *k*-th smallest element in the left part;
if $s < k$-1, look for the (*k*-*s*)-th smallest element in the right part.

Note: The algorithm can simply continue until $s = k$-1

Two Partitioning Algorithms

There are two principal ways to partition an array:

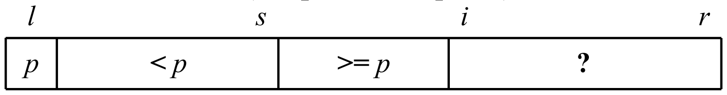
- One-directional scan (Lomuto's partitioning algorithm)

- Two-directional scan (Hoare's partitioning, next lecture)

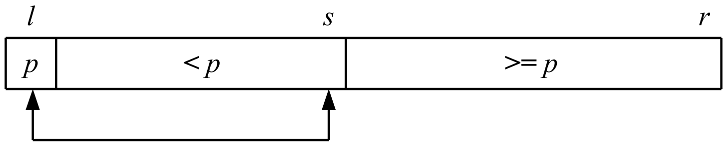
Both algorithms require (*n*-1) key comparisons

Lomuto's Partitioning Algorithm

Scans the array left to right maintaining the array's partition into three contiguous sections: $< p$, $\geq p$, and unknown, where p is the value of the first element (the partition's *pivot*).



On each iteration the unknown section is decreased by one element until it's empty and a partition is achieved by exchanging the pivot with the element in the split position s.



Tracing Lomuto's Partioning Algorithm

s	i							
5	1	10	7	2	6	9	4	15

Tracing Quickselect (Partition-based Algorithm)

Find the median of 4, 1, 10, 9, 7, 12, 8, 2, 15

Here: $n = 9$, $k = \lceil 9/2 \rceil = 5$, $k - 1 = 4$

0	1	2	3	**4**	5	6	7	8
5	1	9	7	2	12	10	4	15
4	1	2	**5**	9	12	10	7	15
				9	12	10	7	15

after 1st partitioning: $s=3<k-1=4$

after 2nd partitioning: $s=$

Worst case complexity:

Efficiency of Quickselect

Average case (average split in the middle):

$C(n) = C(n/2)+(n-1)$ $\qquad$ $C(n) \in \Theta(n)$

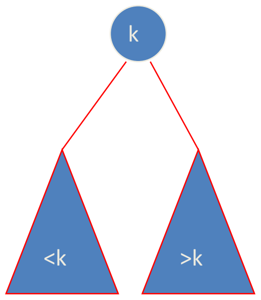
Worst case (degenerate split): $C(n) \in \Theta(n^2)$

A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.

Binary Search Tree Algorithms

Several algorithms on BST requires recursive processing of just one of its subtrees, e.g.,

- Searching
- Insertion of a new key
- Finding the smallest (or the largest) key



Searching in Binary Search Tree

```
Algorithm $BTS(x, v)$
//Searches for node with key equal to $v$ in BST rooted at node $x$
    if $x$ = NIL return -1
    else if $v = K(x)$ return $x$
    else if $v < K(x)$ return $BTS(left(x), v)$
    else return $BTS(right(x), v)$
```

Efficiency

worst case: $C(n) = n$

average case: $C(n) \approx 2\ln n \approx 1.4\log_2 n$

Homework

Read: Ch. 4 and pp. 485-487 of Appendix B

Exercises:

- 4.1: 2, 4, 7, 9, 11
- 4.2: 1, 3, 5, 9
- 4.3: 5, 9a
- 4.4: 2, 4
- 4.5: 2, 7, 13

Next: Divide-and-Conquer (Ch. 5)