CSC 411
Design and Analysis of Algorithms

Chapter 4 Decrease-and-Conquer
- Part 2

Instructor: Minhee Jun
junm@cua.edu

Three major variations of Decrease-and-Conquer

- *Decrease by a constant* (usually by 1):
  - Graph traversal algorithms (DFS and BFS)
  - Insertion sort
  - Topological sorting
  - Algorithms for generating combinatorial objects
- *Decrease by a constant factor* (usually by half)
  - Binary search
  - Fake-Coin Problem
  - Russian Peasant Multiplication
- *Variable size decrease*
  - Computing a Median and the Selection Problem
  - Interpolation Search
  - Searching and Insertion in a Binary Search Tree

4.4 Decrease-by-Constant-Factor Algorithms

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

- Examples:
  - Binary search and the method of bisection
  - Fake-coin puzzle
  - Russian peasant method
  - Josephus problem (Not cover)

4.4.1 Binary Search

- A remarkably efficient algorithm for searching in a sorted array
- Compare a search key K with the array's middle element $A[m]$

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}} . \end{array}$$

Binary Search: Example

Let us apply binary search to searching for K = 70 in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

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: Pseudocode

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
```

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence:

    $C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1$ for $n > 1$, $C_{worst}(1) = 1$

  - For $n = 2^k$,

    $C_{worst}(2^k) = k + 1 = \log_2 n + 1$

  - The solution:

    $C_{worst}(n) = \lfloor log_2 n \rfloor + 1 = \lceil log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_{worst}(n) \in \Theta(\log n)$

Analysis of Binary Search

- 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 $f(x) = 0$.

Exercise 4.4

3. a. What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

b. List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.

4.4.2 Fake-Coin Puzzle

- There are n identically looking coins one of which is fake. Design an efficient algorithm for detecting the fake coin.
  - 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).
  - Assume that the fake coin is known to be lighter than the genuine ones.

Fake-Coin Puzzle

We can easily set up a recurrence relation for the number of weighings $W(n)$ needed by this algorithm in the worst case:

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

This stuff should look elementary by now, if not outright boring.

- But wait: it would be more efficient to divide the coins not into two but into three piles of about $n/3$ coins each.
  - Accordingly, we should expect the number of weighings to be about $\log_3 n$, which is smaller than $\log_2 n$.

General Balance Strategy

- On each step, put $\lceil n/3 \rceil$ of the *n* coins to be searched on each side of the scale.
  - If the scale tips to the left, then:
    - The lightweight fake is in the right set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale tips to the right, then:
    - The lightweight fake is in the left set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale stays balanced, then:
    - The fake is in the remaining set of $n - 2\lceil n/3 \rceil \approx n/3$ coins that were not weighed!
- Except if $n \mod 3 = 1$ then we can do a little better by weighing $\lfloor n/3 \rfloor$ of the coins on each side.

Fake-Coin Puzzle

- Assume that the fake coin is known to be lighter than the genuine ones:
  - Decrease by factor 2 algorithm → $O(\log_2 n)$
  - Decrease by factor 3 algorithm → $O(\log_3 n)$
    - We need w weighing with a balance to find a light counterfeit coin among 3^w coins.
    - So, the number of required weighings with n coins is $w = \lceil \log_3 n \rceil$.

4.4.3 Russian Peasant Multiplication

- The problem:
  - Compute the product of two positive integers n and m

- Can be solved by a **decrease-by-half** algorithm based on the following formulas.
  - For even values of *n*: $n \cdot m = \frac{n}{2} \cdot 2m.$
  - For odd values of n: $n \cdot m = \frac{n-1}{2} \cdot 2m + m.$

Russian Peasant Multiplication: Example

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

n	m	
50	65	
25	130	
12	260	$(+130)$
6	520	
3	1040	
1	2080	$(+1040)$
	2080	$+(130 + 1040) = 3250$

(a)

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

(b)

FIGURE 4.11 Computing $50 \cdot 65$ by the Russian peasant method.

Exercise 4.4

11. **a.** Apply the Russian peasant algorithm to compute $26 \cdot 47$.

b. From the standpoint of time efficiency, does it matter whether we multiply n by m or m by n by the Russian peasant algorithm?

Three major variations of Decrease-and-Conquer

- Decrease by a constant (usually by 1):
  - Graph traversal algorithms (DFS and BFS)
  - Insertion sort
  - Topological sorting
  - Algorithms for generating combinatorial objects
- Decrease by a constant factor (usually by half)
  - Binary search
  - Fake-Coin Problem
  - Russian Peasant Multiplication
- Variable size decrease
  - Computing a Median and the Selection Problem
  - Interpolation Search
  - Searching and Insertion in a Binary Search Tree

4.5 Variable-Size-Decrease Algorithms

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

- Examples:
  - Computing a Median and the Selection Problem
  - Interpolation Search
  - Searching and Insertion in a Binary Search Trees
  - The Game of Nim (Not cover)

4.5.1 Computing Median and Selection Algorithm

- The selection problem
  - finding the *k*-th smallest element in a list of *n* numbers. This number is called the *k*-th **order statistic**.
  - For $k = 1$ or $k = n$, we can simply scan the list in question to find the smallest or largest element, respectively.
  - For $k = n/2$,
    - This middle value is called the **median**, and it is one of the most important notions in mathematical statistics.
    - We can find the *k*-th smallest element in a list by sorting the list first and then selecting the *k*-th element in the output of a sorting algorithm.
    - The time of such an algorithm is determined by the efficiency of the sorting algorithm used.

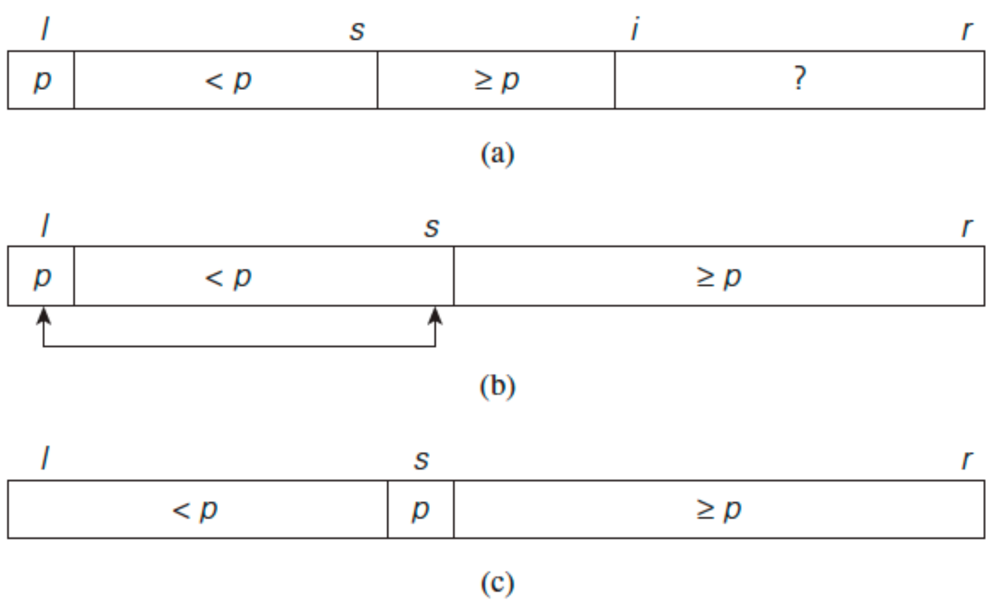
Partition-based Algorithm for the Selection Problem

- Sorting the entire list is most likely overkill since the problem asks not to order the entire list but just to find its 4th smallest element.
- Partition of the list.
  - we can take advantage of the idea of **partitioning** a given list around some value *p* of, say, its first element.

  - In general, this is a rearrangement of the list’s elements so that the left part contains all the elements smaller than or equal to *p*, followed by the **pivot** *p* itself, followed by all the elements greater than or equal to *p*.

Partition-based Algorithm for the Selection Problem

- Assuming that the list is indexed from 1 to *n*:
  - If *p* = *k*, the problem is solved;
  - if p > *k*, look for the *k*-th smallest element in the left part;
  - if *p* < *k*, look for the (*k*-*s*)-th smallest element in the right part.
- The algorithm can simply continue until *s* = *k*.



Pseudocode of the Partition-based Algorithm

ALGORITHM $Quickselect(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
//   integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

Pseudocode of the Partition-based Algorithm

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

EXAMPLE Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the 5th smallest element in the array.

We use the above version of array partitioning, showing the pivots in bold.

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

EXAMPLE Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the 5th smallest element in the array.

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

Efficiency of the Partition-based Algorithm

- Best case (average split in the middle):
  - $C(n) = C(n/2) + n$
  - $C(n) \in \Theta(n)$
- Worst case (degenerate split):
  - $C(n) \in \Theta(n^2)$
- How to avoid the worst case?
  - If we can find a more sophisticated way of choosing a pivot element

Exercise 4.5

2. Apply quickselect to find the median of the list of numbers 9, 12, 5, 17, 20, 30, 8.

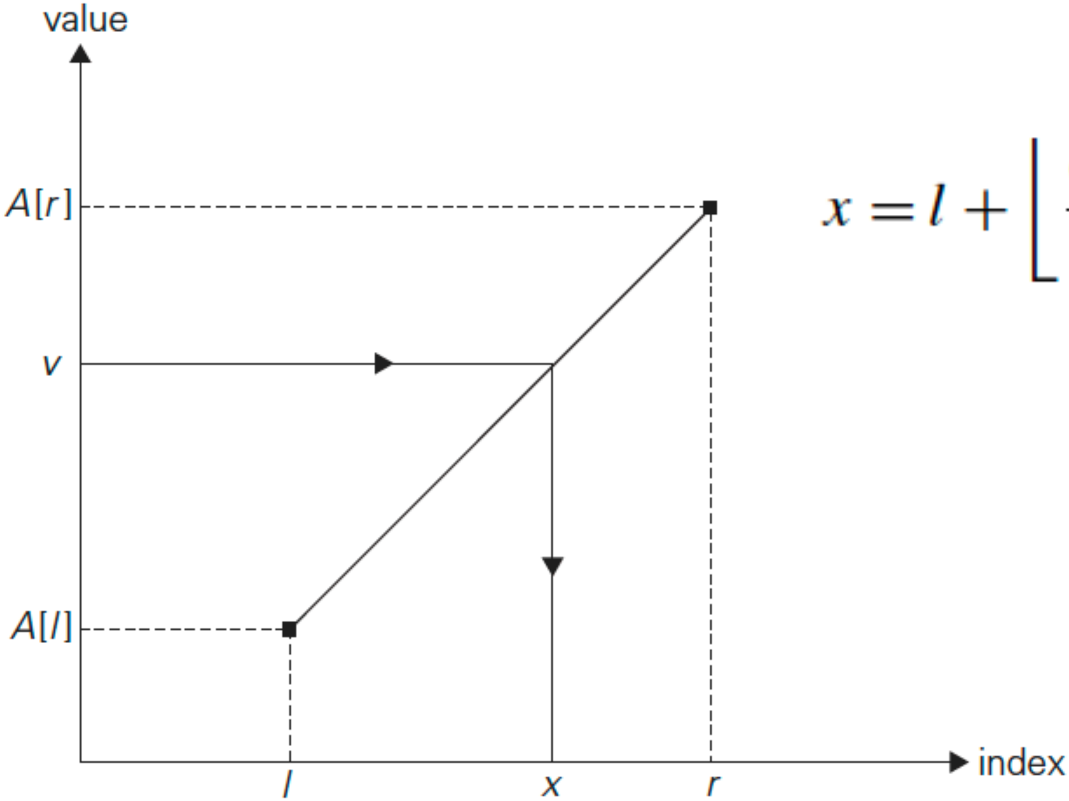
4.5.2 Interpolation Search

- Interpolation search takes into account the value of the search key in order to find the array's element to be compared with the search key.
  - Unlike binary search, which always compares a search key with the **middle value** of a given sorted array.

- Example: a telephone book
  - if we are searching for someone named Brown, we open the book not in the middle but very close to the beginning, unlike our action when searching for someone named, say, Smith.

Interpolation Search

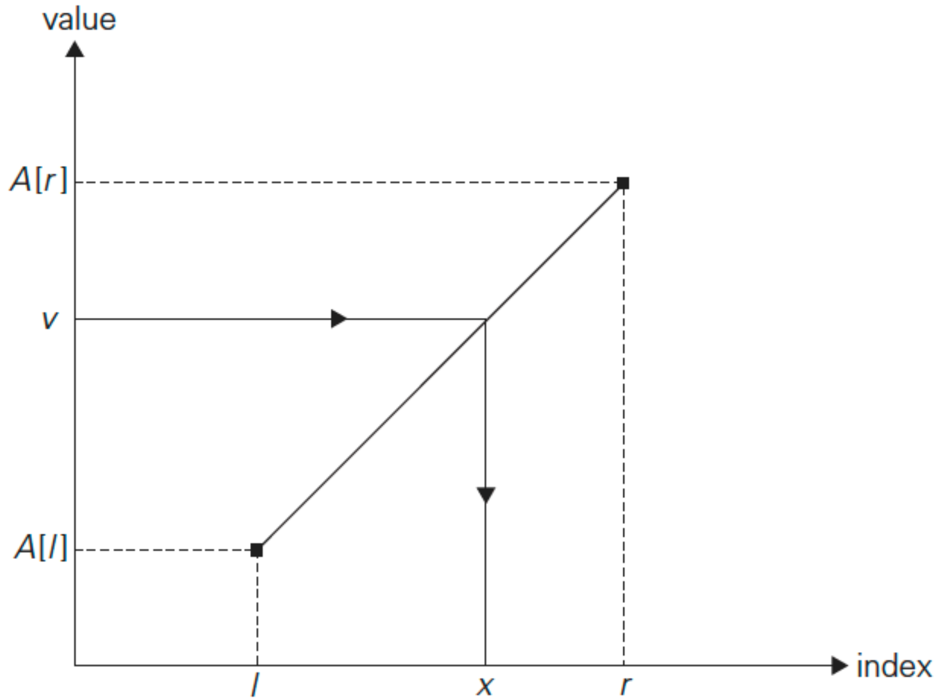
- Searches **a sorted array** similar to binary search but estimates location of the search key in $A[l\,..\,r]$ by using **its value *v*.**



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

Interpolation Search

- After comparing v with $A[x]$,
  - the algorithm either stops (if they are equal) or
  - proceeds by searching in the same manner among the elements indexed either between l and $x - 1$ or between $x + 1$ and r, depending on whether $A[x]$ is smaller or larger than v.



Efficiency of Interpolation Search

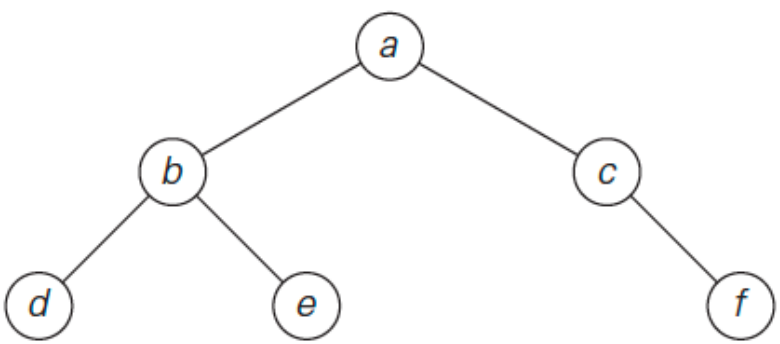
- The size of the problem's instance is reduced, but we cannot tell a priori by how much.
- The analysis of the algorithm's efficiency shows that interpolation search uses fewer than $\log_2 \log_2 n + 1$ key comparisons on the average when searching in a list of n random keys.
- This function grows so slowly that the number of comparisons is a very small constant for all practically feasible inputs (see Problem 6 in this section's exercises).
- But in the worst case, interpolation search is only linear, which must be considered a bad performance (why?).

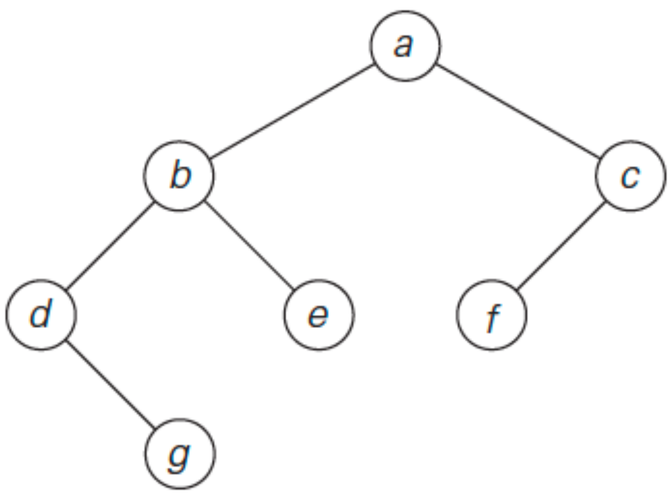
4.5.3 Searching and Insertion in a Binary Search Tree

- Searching for an element of a given value v in such a tree
- Recursively,
  - If the tree is empty, the search ends in failure.
  - Otherwise, we compare v with the tree's root $K(r)$.
    - If they match, a desired element is found and the search can be stopped;
    - Otherwise, we continue with the search
      - in the left subtree of the root, if $v < K(r)$
      - in the right subtree, if $v > K(r)$.
- On each iteration of the algorithm, the problem of searching in a binary search tree is reduced to searching in a smaller binary search tree.

4.5.3 Searching and Insertion in a Binary Search Tree

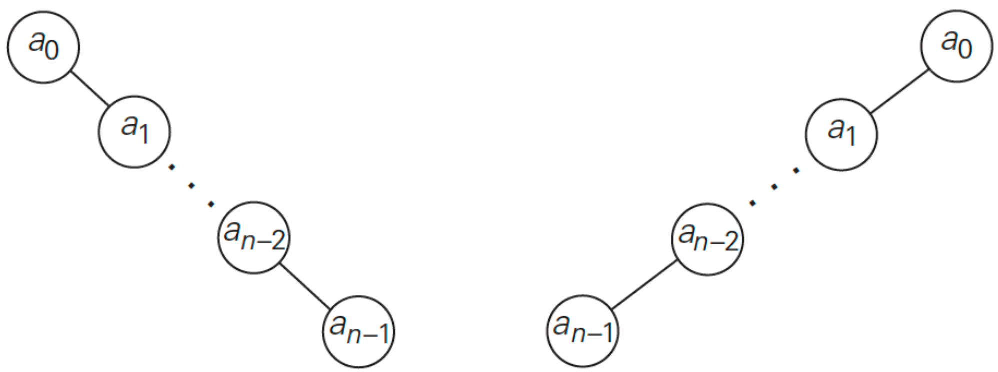
- The most sensible measure of the size of a search tree is its height;
  - the decrease in a tree's height normally changes from one iteration to another of the binary tree search
  - a variable-size-decrease algorithm.





Efficiency: Searching and Insertion in a Binary Search Tree

- Worst case: $\Theta(n)$.
  - A tree is constructed by successive insertions of an increasing or decreasing sequence of keys



(a) (b)

- Average-case efficiency: $\Theta(\log n)$.
  - More precisely, the number of key comparisons needed for a search in a binary search tree is about $2 \ln n \approx 1.39 \log_2 n$.