CS 395 – Analysis of Algorithms

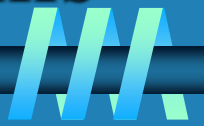
Chapter 4 – Decrease-and-Conquer

Read 4.4 & 4.5

- **Decrease-by-a-Constant-Factor Algorithms**
  - **Binary Search**
  - **Fake Coin Problem**
  - **Russian Peasant Multiplication – Up coming weekly assignment**

- **Variable-Size-Decrease Algorithms**
  - **Median and Selection Problem**
    - **Lomuto Partitioning**
    - **QuickSelect**

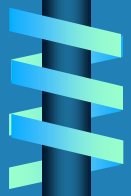
Decrease-by-Constant-Factor Algorithms



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

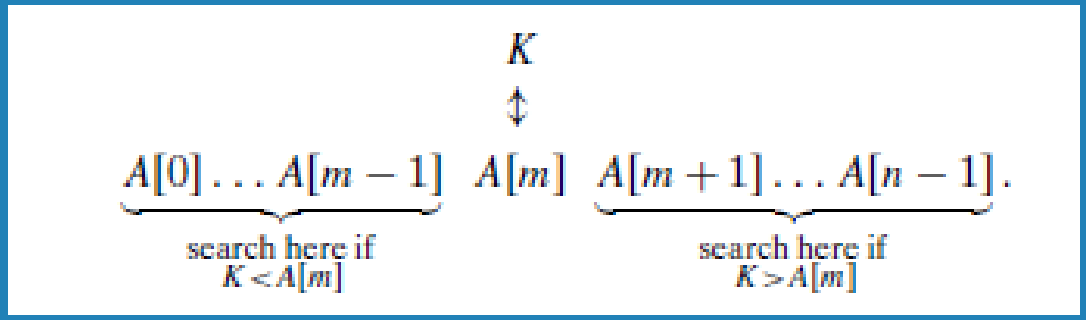
Examples:

- **binary search and the method of bisection**
- **exponentiation by squaring**
- **multiplication à la Russe (Russian peasant method)**
- **fake-coin puzzle**
- **Josephus problem**



Binary Search

Very efficient algorithm for searching in sorted array:

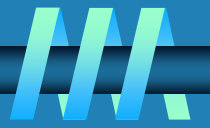


m = middle element of the array

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]

- **Can be implemented both recursively and iteratively**

Binary Search



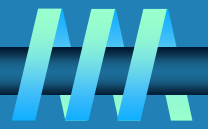
Example:

Let's search for K = 70 in this sorted array

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

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:**
    $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$,
    $C_w(1) = 1$
    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)**

Preview of Master Theorem

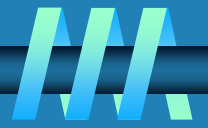
$T(n) = aT(n/b) + n^d$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

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

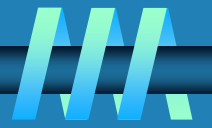
Fake-Coin Puzzle - Pseudocode



Decrease by factor 2 algorithm

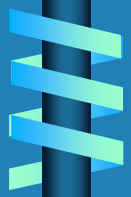
```
ALGORITHM FakeCoin2(A[0..n - 1])
//A decrese-by-factor-2-and-conquer algorithm for the Fake Coin problem
//Input: An array A[0..n - 1]) of n coins
//Output: Identification of the fake coin
if  (n = 1)
    return (the coin is fake)
else   // divide the coins into two piles of n/2 coins each, leaving one extra coin if n is odd
   A1 ← ceiling(n/2)
   A2 ← ceiling(n/2)
   weigh A1 and A2
   if (A1 = A2)
      if (there was one extra coin)
         return (this extra coin is fake)
      else if (A1 < A2)
         FakeCoin2(A1)
      else
         FakeCoin2(A2)
```

Fake-Coin Puzzle - Pseudocode

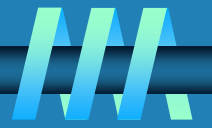


Decrease by factor 3 algorithm

```
ALGORITHM FakeCoin3(A[0..n - 1])
//A decrese-by-factor-3-and-conquer algorithm for the Fake Coin problem
//Input: An array A[0..n - 1]) of n coins
//Output: Identification of the fake coin
if  (n = 1)
    return (the coin is fake)
else   // divide the coins into three piles of n/3 coins each
    A1 ← ceiling(n/3)
    A2 ← ceiling(n/3)
    A3 ← n – (2 * ceiling(n/3))
   weigh A1 and A2
   if (A1 = A2)
      FakeCoin3(A3)
   else if (A1 < A2)
      FakeCoin3(A1)
   else
      FakeCoin3(A2)
```



Analysis of Fake Coin Algorithms



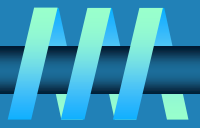
- **Time efficiency**
  - **Factor-2 Algorithm**
    - **Number of weighings *W*(*n*) for Worst-case:**

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

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

  - **Factor-3 Algorithm**
    - ***W*(*n*) = log_3 *n***

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*:

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

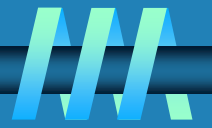
For odd values of *n*:

$$n * m = \frac{n-1}{2} * 2m + m \quad \text{if } n > 1$$

and

$$m \quad \text{if } n = 1$$

Example of Russian Peasant Multiplication



Multiply 50 by 65

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

(a)

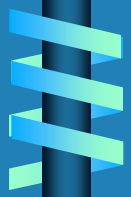
n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

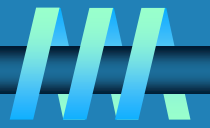
Note: Method reduces to adding *m*'s values corresponding to odd *n*'s.

Pseudocode

ALGORITHM *MultiplyAlaRusse*(n, m)
//Multiplies two integers
//Input: two integers
//Output: multiplication of the input
if $(n = 1)$
    return m
else if $(n\%2 = 0)$
  return *MultiplyAlaRusse* $(n/2, m*2)$
else
    return m + *MultiplyAlaRusse* $(n/2, m*2)$

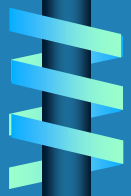


Variable-Size-Decrease Algorithms

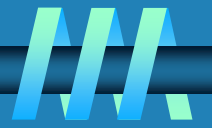


- In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another

- Examples:
  - Euclid's algorithm for greatest common divisor
  - partition-based algorithm for selection problem
  - interpolation search
  - some algorithms on binary search trees
  - Nim and Nim-like games



Selection Problem



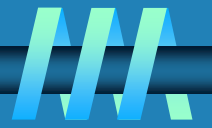
- Find the k-th smallest element in a list of n numbers
  - k = 1 - Smallest element
  - k = n - Largest element
- median: k = $\lceil n/2 \rceil$
- Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = ?
- The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

Algorithms for the Selection Problem

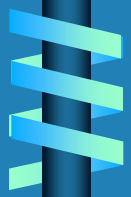
- The sorting-based algorithm: Sort and return the k-th element
  Efficiency (if sorted by Mergesort): Θ(nlog n)
- A faster algorithm is based on the array partitioning:

- 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

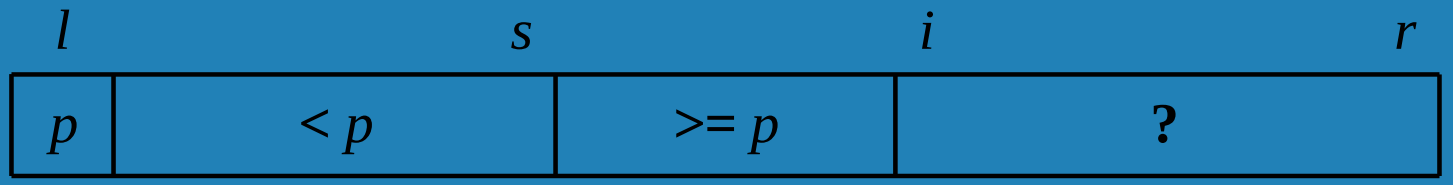


- There are two principal ways to partition an array:
- One-directional scan (Lomuto's partitioning algorithm)
- Two-directional scan (Hoare's partitioning algorithm)

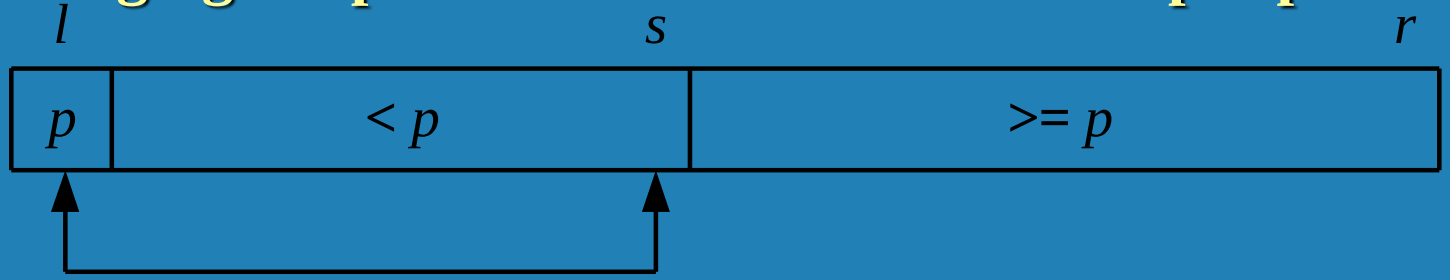


Lomuto's Partitioning Algorithm

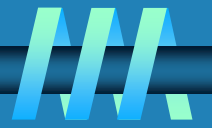
- Scans the array left to right maintaining the array's partition into three contiguous sections: $< p$, $\geq p$, and unknown, where p is the value of the first element (the partition's pivot).



- 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.

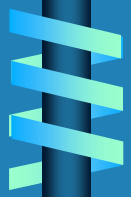


Lomuto's Partitioning Algorithm

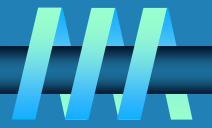


ALGORITHM *LomutoPartition*$(A[l..r])$

//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 $(l \le r)$
//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

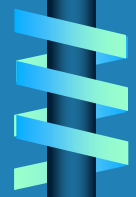


QuickSelect Algorithm

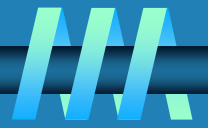


ALGORITHM *Quickselect*$(A[l..r], k)$

//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*$(A[s+1..r], k - 1 - s)$



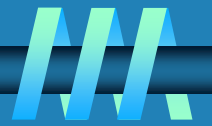
Example



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

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

QuickSelect



- **Time Efficiency**

$$C_{best}(n) = n - 1 \in \Theta(n).$$

$$C_{worst}(n) = (n-1) + (n-2) + \cdots + 1 = (n-1)n/2 \in \Theta(n^2)$$