Decrease-and-Conquer Strategy

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Overview

Decrease-and-Conquer Strategy

Decrease-by-Constant Algorithms

Decrease-by-a-Constant-Factor Algorithms

Variable-Size-Decrease Algorithms

Decrease-and-Conquer Strategy

The **decrease-and-conquer** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.

Three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

Decrease by a Constant

- In the **decrease by a constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm.

Example. The exponentiation problem of computing a^n for positive integer exponents. The function $f(n) = a^n$ can be computed either by "top down" by using its recursive definition

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

or "bottom up" by multiplying a by itself $n-1$ times (iterative).

- The recurrence for investigating the time efficiency of such algorithms has form (typically) $T(n) = T(n-1) + f(n)$

Decrease by a Constant Factor

- The **decrease-by-a-constant factor** technique suggests reducing a problem's instance by the same constant factor on each iteration of the algorithm. Example: binary search algorithm.

Example. The exponentiation problem can be solved using decrease-by-a-constant factor technique by the following formula

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

- The recurrence for investigating the time efficiency of such algorithms has form (typically) $T(n) = T\left(\frac{n}{b}\right) + f(n)$ where $b > 1$.

Variable Size Decrease

- In the **variable-size-decrease** variety of decrease and conquer, a size reduction pattern varies from one iteration of an algorithm to another.

Example. Euclid's algorithm for computing the greatest common divisor which is based on the formula

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

Overview

Decrease-and-Conquer Strategy

Decrease-by-Constant Algorithms

Decrease-by-a-Constant-Factor Algorithms

Variable-Size-Decrease Algorithms

Decrease-by-one Sorting Algorithm

Insertion Sort

- is a direct application of the decrease (by one)-and-conquer technique to sorting an array $A[0 \dots n-1]$
- Assume that smaller instance $A[0 \dots n-2]$ has already been sorted. How can we "solve" for $A[0 \dots n-1]$ in this case?

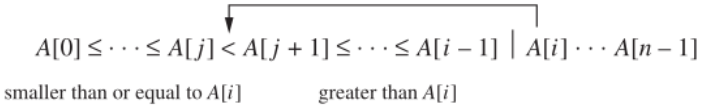


Figure: Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

Insertion Sort

Below is a pseudocode for insertion sort implemented in bottom-up approach

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
```

Insertion Sort

The operation of the algorithm *InsertionSort* is illustrated below.

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

Figure: Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Efficiency Class of Insertion Sort

Best-case Efficiency

- In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop.
- It happens if the input array is already sorted in ascending order.
- For sorted arrays, the number of key comparisons is

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

Efficiency Class of Insertion Sort

Worst-case Efficiency

- In the worst case, $A[j] > v$ is executed the largest number of times, i.e. for every $j = i - 1, \ldots, 0$.
- The worst case input is an array of *strictly* decreasing values.
- The number of key comparisons for such an input is

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

Efficiency Class of Insertion Sort

Average-case Efficiency

- Two-step process
  - (i) Determine ave. number of comparisons needed to move one element into place
  - (ii) Determine overall ave. no. of operations using result of i for all other elements

    Q1: How many positions are there to move the i^{th} element into?

    Q2:: How many comparisons does it take to get to each of these possible positions?
- The average number of key comparisons for a random input is

$$C_{ave}(n) = \sum_{i=1}^{n} \left[\frac{1}{i+1} \left(\sum_{j=1}^{i} j + i \right) \right] \approx \frac{n^2}{4} \in \Theta(n^2).$$

Generating Combinatorial Objects

Generating Permutations

Assume that the underlying set whose elements need to be permuted is simply the set of integers from 1 to n.

- The smaller-by-one problem is to generate all $(n-1)!$ permutations.
- Assuming that the smaller problem is solved, a solution to the larger one is by inserting n in each of the n possible positions among elements of every permutations of $n-1$ elements.

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

Remark

- The same permutation can be done by associating a direction with each element k in a permutation.

  In such an arrow-marked permutation, an element k is said to be **mobile** if its arrow points to a smaller number adjacent to it.

Example

In the permutation $\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}$, the numbers 3 and 4 are mobile while 2 and 1 are not.

Generating Combinatorial Objects

Using the notion of a mobile element, the **Johnson-Trotter algorithm** generates permutations of an $n-$element set.

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Generating Combinatorial Objects

Example

An application of the Johnson-Trotter Algorithm for $n = 3$ with the largest mobile integer shown in bold.

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

Efficiency of Johnson-Trotter Algorithm

Johnson-Trotter algorithm is one of the most efficient for generating permutation and its efficiency class is in $\Theta(n!)$.

Generating Combinatorial Objects

Generating Subsets

- All subsets of $A = \{a_1, \ldots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that contain a_n.
- Once a list of all subsets of $\{a_1, \ldots a_{n-1}\}$ is determined, all the subsets of $\{a_1, \ldots, a_n\}$ is determined by adding to the list all its elements with a_n put into each of them.

Example. Generating all subsets of $\{a_1, a_2, a_3\}$

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

Generating Combinatorial Objects

Procedure

- A convenient way of solving the problem directly is based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1, \ldots, a_n\}$ and all 2^n bit strings $b_1, \ldots, b_n$ of length n.
- Assign to a subset a bit string in which $b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it.

For example, if $n = 3$ we obtain

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

Generating Combinatorial Objects

Squashed Order

An order, in which any subset involving a_j can be listed only after all the subsets involving $a_1 \ldots a_{j-1}$ is called **squashed order**.

Remark

There exists a *minimal-change algorithm* for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit. Such sequence of bit strings is called the **binary reflected Grey code**.

Generating Combinatorial Objects

ALGORITHM $BRGC(n)$

//Generates recursively the binary reflected Gray code of order n
//Input: A positive integer n
//Output: A list of all bit strings of length n composing the Gray code
if $n = 1$ make list L containing bit strings 0 and 1 in this order
else generate list $L1$ of bit strings of size $n - 1$ by calling $BRGC(n - 1)$
    copy list $L1$ to list $L2$ in reversed order
    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 list L
return L

Figure: Pseudocode of Recursive Binary reflected Gray Code

Overview

Decrease-and-Conquer Strategy

Decrease-by-Constant Algorithms

Decrease-by-a-Constant-Factor Algorithms

Variable-Size-Decrease Algorithms

Binary Search

Binary search is an efficient algorithm for searching in a *sorted array*.

- works by comparing a search key K with the array's middle element $A[m]$
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$ and for the second half if $K > A[m]$.

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

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
```

Figure: Pseudocode for the nonrecursive version of the Binary Search

Binary Search

Example. Search for $K = 70$ in the array below using binary search.

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

Efficiency Class of Binary Search

Let $C_{worst}(n)$ be the number of comparisons in the worst-case. Then we have

$$\begin{aligned} C_{worst}(n) &= C_{worst}(\lfloor n/2 \rfloor) + 1 \qquad \text{for } n > 1 \\ C_{worst}(1) &= 1 \end{aligned}$$

The solution of the recurrence relation above for any arbitrary positive integer n is

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

Efficiency Class of Binary Search

For the average-case analysis, we consider two situations: (i) successful search, and (ii) unsuccessful search.

Case i.

- There are n possible locations for the key. Assume that each of these to be equivalent so the probability of each is $1/n$.
- In the binary tree that represents the search process, observe that there are i comparisons done to find elements that are in the nodes on level i.
- There are 2^{i-1} nodes on level i for a binary tree and when $n = 2^k - 1$, there are k levels in the tree.

Efficiency Class of Binary Search

Case i.

- Hence, the average number of comparisons in a successful search given a list of n elements where $n = 2^k - 1$ is

$$\begin{aligned} C_{ave}^{yes}(n) &= \sum_{i=1}^{n} p_i t_i = \frac{1}{n} \sum_{i=1}^{k} i 2^{i-1} = \frac{1}{2n} \sum_{i=1}^{k} i 2^i \\ &= \frac{1}{2n} \left[(k-1)2^{k+1} + 2 \right] = \frac{k2^k - n}{n}. \end{aligned}$$

  Because $n = 2^k - 1$, $2^k = n + 1$ we get $C_{ave}^{yes}(n) = k \left(\frac{n+1}{n} \right) - 1$.

- As n gets larger, $C_{ave}^{yes}(n) \approx k - 1 = \log_2(n+1) - 1 \in \Theta(\log n)$ for $n = 2^k - 1$.

Efficiency Class of Binary Search

Case ii.

- There are n possibilities for the key being in the list, but now we have to add $n+1$ possibilities that key is not in list. In each of the additional $n+1$ possibilities, it takes k comparisons to learn that key is not in list.
- Putting all of this together, we get the average number of comparisons in an unsuccessful search given a list of n elements where $n = 2^k - 1$ is

$$\begin{aligned} C_{ave}^{no}(n) &= \sum_{i=1}^{2n+1} p_i t_i = \frac{1}{2n+1}\left[\sum_{i=1}^{k} i2^{i-1} + \sum_{i=n+1}^{2n+1} k\right] \\ &= \frac{1}{2n+1}\left[(k-1)2^k + 1 + (n+1)k\right]. \end{aligned}$$

Efficiency Class of Binary Search

Case ii.

- After some algebraic manipulation (note: $n = 2^k - 1, 2^k = n + 1$),

$$\begin{aligned} C_{ave}^{no}(n) &= \frac{1}{2^{k+1} - 1}\left[k2^{k+1} - 2^k + 1\right] \\ &\approx \frac{k2^{k+1} - 2^k + 1}{2^{k+1}} \end{aligned}$$

- As n gets larger,

$$C_{ave}^{no}(n) \approx k - \frac{1}{2} = \log_2(n+1) - \frac{1}{2} \in \Theta(\log n)$$

for $n = 2^k - 1$.

Fake-Coin Problem

- Among n identically looking coins, one is fake. With a balance scale, compare any two sets of coins. (*Assume that the fake coin is lighter.*)

  Problem: Design an algorithm for detecting the fake coin.

Algorithm 1

Step 1. Divide n coins into 2 piles of $\lfloor n/2 \rfloor$ coins each. Leave one extra coin aside if n is odd.

Step 2. Put the 2 piles on the scale. If they weigh the same, the coin put aside must be fake; otherwise, proceed to Step 1 with the lighter pile, which must contain the fake coin.

Fake-Coin Problem

Efficiency of Algorithm 1

Let $W(n)$ be the number of weighings needed by Algorithm 1. A recurrence for $W(n)$ in the worst-case is

$$\begin{aligned} W(n) &= W(\lfloor n/2 \rfloor) + 1 \qquad \text{for } n > 1 \\ W(1) &= 0. \end{aligned}$$

Using the Master Theorem, $W(n) \in \Theta(\log_2 n)$.

Try this. Divide the coins into three piles of about $n/3$ coins each. Compare the efficiency of this new algorithm to Algorithm 1.

Russian Peasant Multiplication

- Let n and m be positive integers whose product we want to compute and let the value of n be the measure of instance size.

  (i) If n is even, an instance of half the size has to deal with $n/2$ and the solution to the problem's larger instance is

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

  (ii) If n is odd we have, $n \cdot m = \left(\frac{n-1}{2} \cdot 2m\right) + m$

- Using these formulas and the case of $1 \cdot m = m$ to stop, we can compute the product $n \cdot m$ either *recursively* or *iteratively*.

Overview

Decrease-and-Conquer Strategy

Decrease-by-Constant Algorithms

Decrease-by-a-Constant-Factor Algorithms

Variable-Size-Decrease Algorithms

Variable-Size-Decrease Algorithms

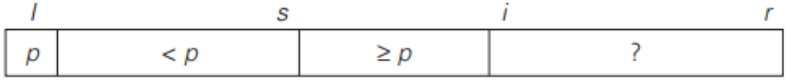
Computing a Median and the Selection Problem

- The **selection problem** is the problem of finding the k^{th} smallest element in a list of n numbers. This number is called the k^{th} **order statistic**.
- A more interesting case of this problem is for $k = \lceil n/2 \rceil$, this middle value is called the **median**.
- Possible algorithms: (1) Sort the list, then return the k^{th} element (algo's efficiency is determined by the efficiency of the sorting algorithm); (2) Take advantage of the idea of **partitioning** the list around some value p (**pivot point**), say its first element, so that left part contains all elements smaller than or equal to p.

Finding the k^{th} Smallest Element

Lomuto Partition Algorithm

1. Consider a subarray $A[l \ldots r]$ of the list $A[0 \ldots n]$. Here, $0 \leq l \leq r \leq n-1$.
2. The subarray $A[l \ldots r]$ composed of three contiguous segments: (i) a segment with elements known to be smaller than p, (ii) the segment of elements known to be greater than or equal to p, and (iii) the segment of elements yet to be compared to p.



3. Starting with $i = l + 1$, scan $A[l \ldots r]$ left to right, until a partition is achieved.

Finding the k^{th} Smallest Element

Lomuto Partition Algorithm (continuation)

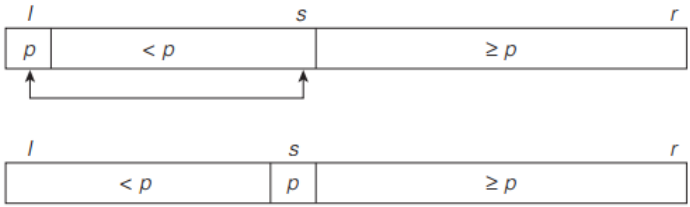
3a. On each iteration in Step 3, compare the first element in the unknown segment with the pivot p.

- If $A[i] \geq p$, i is simply incremented to expand the segment of the elements greater than or equal to p while shrinking the unprocessed segment.
- If $A[i] \leq p$, the segment of the elements smaller than p is expanded by incrementing s, the index of last element in the first segment, swapping $A[i]$ and $A[s]$, and then incrementing i to point to the new first element of the shrunk unprocessed segment.

Finding the k^{th} Smallest Element

Lomuto Partition Algorithm (continuation)

3b. After no unprocessed elements remain, the algorithm swaps the pivot p with $A[s]$ to achieve a partition being sought.



Lomuto Partition 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 ≤ r)
//Output: Partition of A[l..r] and the new position of the pivot
p ← A[l]
s ← l
for i ← l + 1 to r do
    if A[i] < p
        s ← s + 1;   swap(A[s], A[i])
swap(A[l], A[s])
return s
```

Figure: Pseudocode of the partitioning procedure detailed in Lomuto Partition Algo.

Finding k^{th} Smallest Element

Quickselect Algorithm

- Consider the partition: $\underbrace{a_0 \dots a_{s-1}}_{\leq p}\ p\ \underbrace{a_{s+1} \dots a_n}_{\geq p}$ of the list $A[0 \dots n-1]$ using Lomuto Partitioning algorithm with some element in list as pivot point p. Here, s is the partition's split position, i.e., the index of the array's element occupied by the pivot after partitioning.

  (i) If $s = k-1$, the pivot p is the k^{th} smallest element.

  (ii) If $s > k-1$, the k^{th} smallest element can be found as the k^{th} smallest element in the left part of the partitioned array.

  (iii) If $s < k-1$, proceed by searching for the $(k-s)^{th}$ smallest element in the right part of the partitioned array.

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

Figure: Pseudocode of the Quickselect algorithm, which finds the k^{th} element in $A[l \ldots r]$

Finding the k^{th} Smallest Element

Efficiency of the Quickselect Algorithm

- Partitioning an $n-$element array requires $n-1$ key comparisons.
- If the partitioning algorithm produces a split that solves the selection problem without requiring more iterations, then $C_{best}(n) = n-1 \in \Theta(n)$.
- The partitioning algorithm can produce an extremely unbalanced partition, with one part being empty and the other containing $n-1$ elements. In the worst case, this can happen on each of the $n-1$ iterations. For example, if $A[0 \ldots n-1]$ is sorted and you are looking for the largest element, then

$$C_{worst}(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2} \in \Theta(n^2).$$

Finding the k^{th} Smallest Element

Remarks

- The **median selection algorithm** is a special case of the Quickselect (take $k = \lceil n/2 \rceil$).
- A careful mathematical analysis of the quickselect algorithm has shown that $C_{ave}(n) \in \Theta(n)$ (*average-case efficiency is linear*).
- Computer scientists have discovered a more sophisticated way of **choosing a pivot** in quickselect that guarantees *linear time even in the worst case*, however, is too complicated to be recommended for practical applications.
- *Generality of the Algorithm.* The partition-based algorithm solves a somewhat more general problem of identifying the k smallest and $n - k$ largest elements of a given list, not just the value of its k^{th} smallest element.

Variable-Size-Decrease Algorithms

Searching and Insertion in a Binary Search Tree (BST)

- A BST is a tree whose nodes contain elements of a set of orderable items so that for every node all elements in left subtree are smaller and all elements in the right subtree are greater than the element in the subtree's root.
- Searching an element v in a BST *recursively*:

  (i) If tree is empty, search ends.

  (ii) Otherwise, compare v with tree's root $K(r)$. If $v = K(r)$, done; if $v < K(r)$, search in the left subtree; if $v > K(r)$, search in right subtree.

On each iteration, problem of searching in a BST is reduced to searching in a smaller BST. The worst case occurs if tree is severely skewed (e.g. tree constructed by successive insertions of increasing/decreasing sequence of keys).

End of Lecture