CS-203 – WEEK 04 – DECREASE-AND-CONQUER

GIUSEPPE TURINI

Kettering
UNIVERSITY

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- * "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 4, pp. 131-168.

Practice Exercises:

- Exercises from *: 4.1.1-2, 4.1.4-5, 4.1.7-11, 4.2.1-8, 4.3.1-10, 4.4.1-5, 4.4.9, 4.5.2-8, 4.5.13.

Additional Resources:

- VisuAlgo

INTRODUCTION TO DECREASE-AND-CONQUER

The decrease-and-conquer is an algorithm design technique that is based on the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.

Once this relationship is established, it can be exploited top-down or bottom-up:

- Top-down leads naturally to a recursive implementation (e.g., P(n)=P(n-1)+1).
- Bottom-up is usually implemented iteratively (e.g., P(0) → P(1) → ... → P(n)).

There are three major variations of decrease-and-conquer:

- Decrease-by-a-constant (e.g., decrease-by-1, from P(n) to P(n-1)).
- Decrease-by-a-constant-factor (e.g., decrease-by-factor-2, from P(n) to P(n/2)).
- Variable-size-decrease (e.g., from P(n) to P(n-a) with a variable).

EXPONENTIATION BY BRUTE-FORCE: ALGORITHM

Now consider the simple problem of Exponentiation (or computing the power function).
We will solve this problem using multiple algorithms with different designs.

Problem: Compute a^n (exponentiation or power function) with $a \neq 0$ and $n \geq 0$.

Exponentiation (Power Function) by Brute Force
The straightforward way to solve an instance of size n (a^n) is to compute a^n performing n-1 multiplications 1-by-1. So, the function pow(a, n) = a^n, can be computed iteratively (incremental approach, or bottom-up) using this formula:

$$\text{exponentiation} = a^n = \text{pow}(a, n) = a \times \cdots \times a, \quad n - 1 \text{ multiplications.}$$

Note: This algorithm can also be considered a decrease-by-1 algorithm implemented bottom-up (iteratively) if you consider that pow(a, n) = $a^n = a^{n-1} \times a = \ldots = a^1 \times \ldots \times a$

EXPONENTIATION BY BRUTE-FORCE: IMPLEMENTATION

This is a Java implementation of the Exponentiation by Brute-Force algorithm (version 1 not-optimized, version 2 optimized).

```
public static int Pow1( int a, int n ) { // Compute exponentiation.
   int res = 1; // Init temp result.
   while( n > 0 ) { // Iterate multiplications until exponent is exhausted.
      res = res * a; // Perform multiplication by base.
      n--; // Decrement current exponent.
   }
   return res; // Return result.
}

public static int Pow2( int a, int n ) {
   if( n == 0 ) { return 1; } // Check special cases (n==0).
   else if ( n == 1 ) { return a; } // Check special cases (n==1).
   else { // Compute exponentiation (n>1).
      int res = a; // Init temp result.
      while( n > 1 ) { // Iterate multiplications until exponent is exhausted.
         res = res * a; // Perform multiplication by base.
         n--; // Decrement current exponent.
      }
      return res; // Return result.
   }
}
```

EXPONENTIATION BY BRUTE-FORCE: ANALYSIS

The input size is the magnitude of the exponent: the greater the exponent the more work the algorithm has to do, and the base is not affecting the work done by the algorithm at all.
The basic operation is the multiplication.
The basic operation count C(n) depends only on the input size (so no need of best-, worst-, and average- case analyses).

$$\text{basic operation count version } 1 = \ C_1(n) = \sum_{i=1}^{n} 1 = n \in \Theta(n)$$

$$\text{basic operation count version } 2 = \ C_2(n) = \sum_{i=2}^{n} 1 = n - 1 \in \Theta(n)$$

Note: Version 2 is slightly faster than version 1 (1 multiplication less); however, this optimization is not enough to improve the efficiency class of the algorithm

DECREASE-BY-A-CONSTANT

In the decrease-by-a-constant design, the size of an instance is reduced by the same constant value on each iteration of the algorithm (e.g., from P(n) to P(n-1)).
Usually, the constant decrease is by 1, but other constant size reductions exist (see Figure 1).

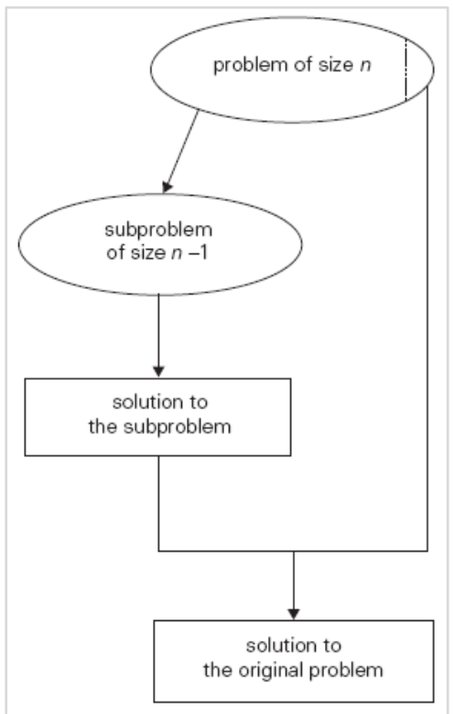

Figure 1. Flowchart illustrating the decrease-by-1 algorithm design technique.

EXPONENTIATION BY DECREASE-BY-1: ALGORITHM

Problem: Compute a^n (exponentiation or power function) with $a \neq 0$ and $n \geq 0$.

Exponentiation (Power Function) Using Decrease-by-1

The relationship between a solution to an instance of size n (a^n) and an instance of size n-1 (a^{n-1}) is obtained by the formula $a^n = a^{n-1} \times a$. So, the function pow(a, n) = a^n, can be computed recursively (top-down) using these recursive definitions (version 3 not-optimized, version 4 optimized):

$$\text{exponentiation (original)} = a^n = \text{pow}_3(a, n) = \begin{cases} 1, & \text{if } n = 0. \\ \text{pow}_3(a, n-1) \times a, & \text{if } n > 0. \end{cases}$$

$$\text{exponentiation (optimized)} = a^n = \text{pow}_4(a, n) = \begin{cases} 1, & \text{if } n = 0. \\ a, & \text{if } n = 1. \\ \text{pow}_4(a, n-1) \times a, & \text{if } n > 1. \end{cases}$$

EXPONENTIATION BY DECREASE-BY-1: IMPLEMENTATION

This is a Java implementation of the Exponentiation by Decrease-by-1 algorithm (version 3 not-optimized, version 4 optimized).

```
public static int Pow3Rec( int a, int n ) {
   // BASE CASE: check if exponent is 0.
   if( n == 0 ) { return 1; }
   else {
      // RECURRENCE: recursive call using same base but smaller exponent.
      return a * Pow3Rec(a, n-1);
   }
}

public static int Pow4Rec( int a, int n ) {
   // BASE CASES: check if exponent is 0 or 1.
   if( n == 0 ) { return 1; }
   else if( n == 1 ) { return a; }
   else {
      // RECURRENCE: recursive call using same base but smaller exponent.
      return a * Pow4Rec(a, n-1);
   }
}
```

EXPONENTIATION BY DECREASE-BY-1: ANALYSIS

The input size is the magnitude of the exponent: the greater the exponent the more work the algorithm has to do, and the base is not affecting the work done by the algorithm at all.
The basic operation is the multiplication.
The basic operation count $C_4(n)$ (version 4, optimized) depends only on the input size (so no need of best-, worst-, and average- case analyses).

The Exponentiation by decrease-by-1 is a recursive algorithm; so, at first, we have to define the count of basic operations as a recursive definition.

$$\text{count of basic operations (multiplications)} = C_4(n) = \begin{cases} 0, & \text{if } n = 0. \\ 0, & \text{if } n = 1. \\ C_4(n-1) + 1, & \text{if } n > 1. \end{cases}$$

Note: Here we approximated the work done in the base cases (n=0, n=1) as 0 even if it was very small (but not 0). In this case this approach was correct; but, in general, be careful in estimating small constant work as 0 (it is better to use a constant >0).

EXPONENTIATION BY DECREASE-BY-1: ANALYSIS 2

Now, we can convert the recursive definition of the count of basic operations $C_4(n)$ (see previous slide) in a closed-form formula.

In this case, we use the Backward Substitutions.

$$C_4(n) = C_4(n-1) + 1 =$$

apply recurrence

$$= [\, C_4(n-2) + 1 \,] + 1 = C_4(n-2) + 2 =$$
$$= [\, C_4(n-3) + 1 \,] + 2 = C_4(n-3) + 3$$

$$\Downarrow$$

create pattern $\quad C_4(n) = C_4(n-i) + i, \quad \text{with } i \in [0, n-1]$

$$\Downarrow$$

solve pattern $\quad C_4(n-i) + i = C_4(n-(n-1)) + n - 1 = C_4(1) + n - 1 = n - 1$

basic operation count (optimized version) $= \; C_4(n) = n - 1 \in \Theta(n)$

INSERTION SORT: ALGORITHM

Problem: Sort a list of n orderable items, rearranging them in non-decreasing order.

Insertion Sort A decrease-by-1 algorithm based on the idea that we can sort and array with n items, assuming that a smaller array with n-1 items is already sorted.

So, if A is an array with n items and we assume its first n-1 items are already sorted, then the problem is to correctly reposition the last item (A[n-1]).

Step 1: Scan sorted subarray right-to-left until find item $\leqslant$ A[n-1].

Step 2: Insert A[n-1] right after that item.

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 2. Insertion Sort of list { 89, 45, 68, 90, 29, 34, 17 }. Each row is 1 pass of the algorithm. Separator splits the list between its sorted/unsorted partitions.

INSERTION SORT: IMPLEMENTATION

This is a Java implementation of Insertion Sort (iterative, bottom-up).

```
public static void InsertionSort( int[] A ) {
   int val = 0; // Init temp variable.
   int j = 0; // Init temp variable.
   for( int i = 1; i < A.length; i++ ) { // Iterate array items from 2nd to last.
      val = A[i]; // Update temp variable.
      j = i-1; // Update temp variable.
      // Scan array from before-curr to first, right-shifting when needed.
      while( ( j >= 0 ) && ( A[j] > val ) ) {
         A[j+1] = A[j]; // Right shift to order items.
         j = j-1; // Decrement index for right-to-left scan.
      }
      A[j+1] = val; // Store current item at proper cell after right shift.
   }
}
```

Note: Unlike in Selection Sort, in Insertion Sort the i items already sorted are generally not in their final sorted positions.

INSERTION SORT: ANALYSIS

The input size is the item-size of the list, and the basic operation is the comparison.
The basic operation count C(n) depends on input content, so we study best-/worst- cases.

- The **worst-case** happens for arrays of strictly decreasing values.

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

- The **best-case** happens for arrays already sorted.

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

- So, the general count C(n) is in: $C(n) \in O(n^2)$ and $C(n) \in \Omega(n)$

LOMUTO PARTITIONING: ALGORITHM

Problem: Given an input array A and an array item p, partition A around p (rearrange A so that its left part contains all items $\leqslant$p, then the pivot p, followed by all the items $\geqslant$p).

Lomuto Partitioning (Using Decrease-by-1)

Consider the array A (or, more generally, a subarray A[left...right] with 0$\leqslant$left$\leqslant$right$\leqslant$n-1) as composed of 4 contiguous segments (listed below from left to right, see Figure 3):

1: The pivot p (stored at index left of the subarray by default).
2: Segment <p (storing items <p), delimited by indices left+1 and s.
3: Segment $\geqslant$p (storing items $\geqslant$p), delimited by indices s+1 and i-1.
4: Unknown segment (items not processed yet), delimited by indices l and right.

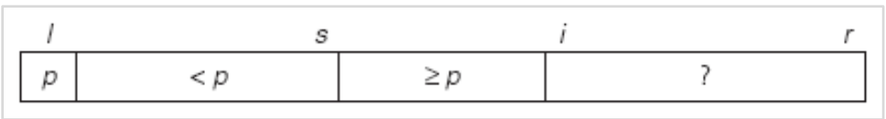

Figure 3. Array segments used by the Lomuto Partitioning algorithm.

LOMUTO PARTITIONING: ALGORITHM 2

The Lomuto Partitioning algorithm scans the subarray A[left...right] from left to right, keeping the segments-structure (see Figure 4) until a partition is achieved. On each iteration:

Step 1: Compare the 1st item in the unknown segment (at index i) with the pivot p.

Step 2: If A[i] ⩾ p, increment i (expand segment ⩾p, shrink unknown segment).

Step 3: If A[i] < p, increment s (expand segment <p), swap A[i] A[s], then increment i.

Step 4: When unknown segment is empty, swap A[left] (pivot) with A[s].

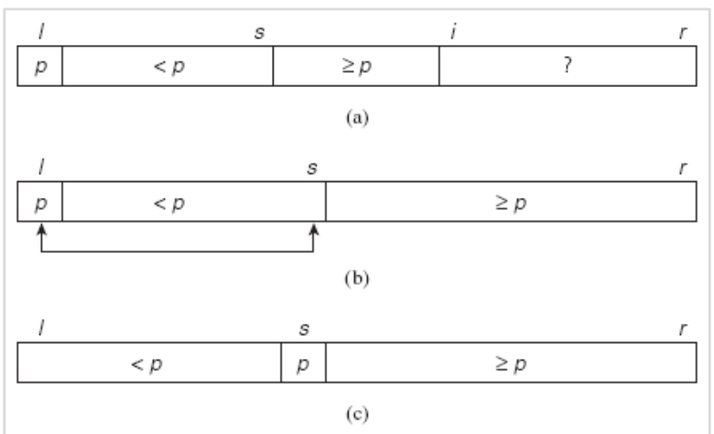

Figure 4. Illustration of the Lomuto Partitioning algorithm: (a) the array segments used by the algorithm, (b) the final swap to reposition the pivot (Step 4), and (c) the final array partitioning.

LOMUTO PARTITIONING: IMPLEMENTATION

This is a Java implementation of the Lomuto Partitioning by Decrease-by-1(iterative).

```
public static int LomutoPartitioning( int[] A, int left, int right ) {
   // Init internal variables.
   int p = A[left]; // Pivot value, initialized as first item in subarray.
   int s = left; // s, last index of segment <p.
   // Process all items in unknown segment.
   for( int i = left+1; i <= right; i++ ) {
      // Compare current unprocessed item (at index i) with pivot (p).
      if( A[i] < p ) {
         s++; // Increment s to expand segment <p.
         // Swap A[s] with A[i], to reposition item at index i to index s.
         int temp = A[s]; A[s] = A[i]; A[i] = temp;
      }
      // Note: in any case index i is incremented.
   }
   // Swap A[left] with A[s], to reposition pivot (p) at index left to index s.
   int temp = A[left]; A[left] = A[s]; A[s] = temp;
   // Return index of pivot in final partitioning.
   return s;
}
```

LOMUTO PARTITIONING: ANALYSIS

The input size is the item-size (n) of the input subarray (from left to right, inclusive).
So, n = right – left +1.

The basic operation is the comparison (array item vs. pivot).

The basic operation count C(n) depends only on the input size (no need of best-/worst- cases).

$$\text{basic operation count} = \mathrm{C}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

HOARE PARTITIONING: ALGORITHM

Problem: Given an input array A and an array item p, partition A around p (rearrange A so that its left part contains all items $\leqslant$p, then the pivot p, followed by all the items $\geqslant$p).

Hoare Partitioning (Using Decrease-by-1)

We start by selecting a pivot. For simplicity, we select the 1st item in the subarray (from index left to index right) as the pivot p (p=A[left]). Then we scan the subarray from both ends (double-scans, left-to-right and right-to-left), comparing items to the pivot:

- **Left-to-right scan (index i):** starts from the 2nd item of the subarray. We want items smaller than the pivot in the left segment, so this scan skips over items smaller than the pivot, and stops when the 1st item not smaller than the pivot is encountered.
- **Right-to-left scan (index j):** starts from the last item of the subarray. We want items greater than the pivot in the right segment, so this scan skips over items greater than the pivot, and stops when the 1st item not greater than the pivot is encountered.

HOARE PARTITIONING: ALGORITHM 2

In running the Hoare Partitioning, whenever both scans stop:

- **If scan indices have not crossed (i<j):** we swap A[i] and A[j], and resume both scans.
- **If scan indices have crossed (i>j):** double-scans end, and we complete the partitioning by swapping pivot (at index left) with last item in smaller segment (at index j).
- **If scan indices have met (i=j):** the value they point is equal to the pivot, so the partitioning is complete.

HOARE PARTITIONING: IMPLEMENTATION

This is a Java implementation of the Hoare Partitioning by Decrease-by-1(iterative).

```
public static int HoarePartitioning( int[] A, int left, int right ) {
   // Pivot selected as the first element in input array.
   int pivotIndex = left;
   int pivotValue = A[pivotIndex];
   // Init internal indices for double-scans (see do-while).
   int i = left - 1;
   int j = right + 1;
   // Loop to perform multiple double-scans.
   while(true) {
      // Left-to-right scan.
      do { i++; } while ( A[i] < pivotValue );
      // Right-to-left scan.
      do { j--; } while ( A[j] > pivotValue );
      // If scan indices cross, return; otherwise, swap A[i] with A[j].
      if( i < j ) {
         int temp = A[i]; A[i] = A[j]; A[j] = temp;
      }
      else { return j; } // Warning: j is index of last item in left part (<pivot)!
   }
}
```

HOARE PARTITIONING: ANALYSIS

The input size is the item-size (n) of the input subarray (from left to right, inclusive).
So, n = right – left +1.

The basic operation is the comparison (array item vs. pivot).

The basic operation count C(n) depends only on the input size (no need of best-/worst- cases).

$$\text{basic operation count} = \mathrm{C}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

COMPARISON: LOMUTO VS HOARE PARTITIONING

Both Lomuto Partitioning and Hoare Partitioning perform the same number of comparisons.

Lomuto Partitioning needs ~3 times the number of swaps than Hoare Partitioning.

Lomuto Partitioning is semi-stable.

Hoare Partitioning is unstable.

Lomuto Partitioning can be used with any linear data structure (array, linked lists, etc.).

Hoare Partitioning needs a bidirectional data structure (array, doubly linked list, etc.).

DECREASE-BY-A-CONSTANT-FACTOR

In the decrease-by-a-constant-factor design, the size of an instance is reduced by the same constant factor on each iteration of the algorithm (e.g., from P(n) to P(n/2)).
Usually, the constant factor is 2, but other constant factor reductions exist (see Figure 5).

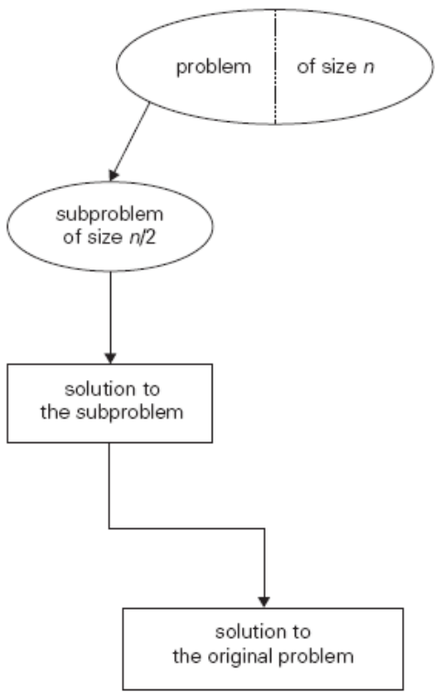

Figure 5. Flowchart showing the decrease-by-factor-2 algorithm design technique.

EXPONENTIATION BY DECREASE-BY-FACTOR-2: ALGORITHM

Problem: Compute a^n (exponentiation or power function) with $a \neq 0$ and $n \geq 0$.

Exponentiation (Power Function) Using Decrease-by-Factor-2 *

The relationship between a solution to an instance of size n (a^n) and an instance of size n/2 ($a^{n/2}$) is obtained by the formula $a^n = a^{n/2} \times a^{n/2}$. So, the function pow(a, n) = a^n, can be computed recursively (top-down) using this recursive definition:

$$\text{exp.} = a^n = \text{pow}(a, n) = \begin{cases} \text{pow}\left(a, \frac{n}{2}\right) \times \text{pow}\left(a, \frac{n}{2}\right), & \text{if } n > 0 \text{ and } n \text{ is even.} \\ \text{pow}\left(a, \left\lfloor \frac{n}{2} \right\rfloor\right) \times \text{pow}\left(a, \left\lfloor \frac{n}{2} \right\rfloor\right) \times a, & \text{if } n > 0 \text{ and } n \text{ is odd.} \\ 1, & \text{if } n = 0. \end{cases}$$

*: This strategy (not-optimized version) is actually a Divide-and-Conquer strategy; but, the optimization performed makes it a Decrease-by-Factor-2 algorithm (optimized version).

EXPONENTIATION BY DECREASE-BY-FACTOR-2: IMPLEMENTATION

This is a Java implementation of the Exponentiation by Decrease-by-Factor-2 algorithm (version 5 not-optimized, version 6 optimized).

```
public static int Pow5Rec( int a, int n ) {
   if( n == 0 ) { return 1; } // BASE CASE: check if exponent is 0.
   else {
      if( ( n % 2 ) == 0 ) { return Pow5Rec(a, n/2) * Pow5Rec(a, n/2); }
      else { return Pow5Rec(a, n/2) * Pow5Rec(a, n/2) * a; }
   }
}

public static int Pow6Rec( int a, int n ) {
   if( n == 0 ) { return 1; } // BASE CASES: check if exponent is 0 or 1.
   else if( n == 1 ) { return a; }
   else {
      int tmpRes = Pow6Rec(a, n/2);
      if( ( n % 2 ) == 0 ) { return tmpRes * tmpRes; }
      else { return tmpRes * tmpRes * a; }
   }
}
```

Note: This code relies on the integer division integrating the rounding down (e.g., 3/2 = 1).

EXPONENTIATION BY DECREASE-BY-FACTOR-2: ANALYSIS

The input size is the magnitude of the exponent: the greater the exponent the more work the algorithm has to do, and the base is not affecting the work done by the algorithm at all.
The basic operation is the multiplication.
The basic operation count $C_6(n)$ (version 6, optimized) depends only on the input size (so no need of best-, worst-, and average- case analyses).

The Exponentiation by decrease-by-1 is a recursive algorithm; so, at first, we have to define the count of basic operations as a recursive definition.

$$\text{basic operation count} = C_6(n) = \begin{cases} 0, & \text{if } n = 0. \\ 0, & \text{if } n = 1. \\ C_6(\,n/2\,) + 1, & \text{if } n > 1 \text{ and } n \text{ is even.} \\ C_6(\,\lfloor n/2 \rfloor\,) + 2, & \text{if } n > 1 \text{ and } n \text{ is odd.} \end{cases}$$

Note: The code relied on the integer division integrating the rounding down (e.g., 3/2 = 1), but this recursive definition has to explicitly include the flooring when needed.

EXPONENTIATION BY DECREASE-BY-FACTOR-2: ANALYSIS 2

Now, we can convert the recursive definition of the basic operation count $C_6(n)$ (see previous slide) in a closed-form formula.

In this case, we use the Backward Substitutions together with the Smoothness Rule.

The Smoothness Rule (under specific conditions) allows us to consider only n that are powers of b (e.g., with b = 2 we have $n = 2^k$ or $k = \log_2 n$) and then extend our result to the general case.

$$\text{apply recurrence} \quad C_6(n) = C_6(2^k) = C_6(n/2) + 1 =$$
$$= [\, C_6((n/2)/2) + 1 \,] + 1 = C_6(n/4) + 2 =$$
$$= [\, C_6((n/4)/2) + 1 \,] + 2 = C_6(n/8) + 3$$

$$\Downarrow$$

$$\text{create pattern} \quad C_6(n) = C_6(2^k) = C_6(n/2^i) + i, \quad \text{with } i \in [0, k]$$

$$\Downarrow$$

$$\text{solve pattern} \quad C_6(n/2^i) + i = C_6(n/2^k) + k = C_6(2^k/2^k) + k = C_6(1) + k = \log_2 n$$

EXPONENTIATION BY DECREASE-BY-FACTOR-2: ANALYSIS 3

So, we have converted the recursive definition of the basic operation count $C_6(n)$ in a closed-form formula, by using the Backward Substitutions with the Smoothness Rule ($n = 2^k$).

$$\text{basic operation count (optimized version)} = C_6(n) = \log_2 n \in \Theta(\log n)$$

Note: $C_6(n) = \log_2 n$ is in the class of efficiency $\Theta(\log n)$. Using the logarithm base in the efficiency class is meaningless, because switching among logarithms of different bases only requires a constant multiplicative factor (e.g., $\log_2 n \times \log_5 2 = \log_5 n$).

Note: Please consider the differences in time efficiency of all these different algorithms for the Exponentiation problem:

- The least efficient, performing n multiplications, in class $\Theta(n)$.
- The least efficient optimized, performing n-1 multiplications, in class $\Theta(n)$.
- The most efficient, performing $\log_2 n$ multiplications, in class $\Theta(\log n)$.

BINARY SEARCH: ALGORITHM

Problem: Given an input item, search it in a sorted array, returning its position (if found).

Binary Search (Using Decrease-by-Factor-2)

This is an efficient algorithm to search for a given item (k) in a sorted array (A). It works by:

Step 1: Compare the search key k with the middle element A[m] of array A.

Step 2: Check if k and A[m] match: if so, stop and return m; otherwise, go to Step 3.

Step 3: Repeat Step 1, limiting search to 1st half of A (if k < A[m]), or limiting search to 2nd half of A (if k > A[m]).

Step 4: The algorithm stops when the search key k is found (success), or when the search partition becomes empty (failure).

BINARY SEARCH: ALGORITHM 2

Example: Table 1 shows an application of the Binary Search algorithm to search for the key k=70 in a sorted array with 13 items.

Table 1. An example of 3 iterations of the Binary Search algorithm to search for item 70 (k) on a sorted array with 13 items: highlighted the shrinking of the search partition (from index L to index R, inclusive) during Binary Search steps.

Array Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Array 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				

Note: The Binary Search algorithm (designed for the discrete domain) has a twin for the continuous domain called Bisection Method (solving equations in 1 unknown, f(x)=0).

BINARY SEARCH: IMPLEMENTATION

This is a Java implementation of the Binary Search by Decrease-by-Factor-2 algorithm (version 1, iterative top-down).

```
public static int BinarySearch1( int[] A, int k ) {
   // Init temp variables (search partition).
   int left = 0;
   int middle = 0;
   int right = A.length - 1;
   // Search until range (search partition) is invalid.
   while( left <= right ) {
      middle = (int) Math.floor( ( left + right ) / 2 ); // Update pivot (middle).
      if( k == A[middle] ) { return middle; } // Key found at pivot (middle).
      else if( k < A[middle] ) { right = middle - 1; } // Update range (left half).
      else { left = middle + 1; } // Update range (right half).
   }
   return -1; // Loop ended without finding input key: search failed, return -1.
}
```

BINARY SEARCH: IMPLEMENTATION 2

This is a Java implementation of the Binary Search by Decrease-by-Factor-2 algorithm (version 2, recursive top-down).

```
public static int BinarySearch2Rec( int[] A, int k, int left, int right ) {
   // BASE CASE: check if search partition is empty
   if( left > right ) { return -1; }
   else {
      // Search partition has at least 1 element, perform binary search.
      // Update pivot (middle).
      int middle = (int) Math.floor( ( left + right ) / 2 );
      if( k == A[middle] ) { return middle; } // Key found at pivot (middle).
      else if( k < A[middle] ) {
         // Proceed search (recursively) in left half.
         return BinarySearch2Rec( A, k, left, middle - 1 ); }
      else {
         // Proceed search (recursively) in right half.
         return BinarySearch2Rec( A, k, middle + 1, right ); }
   }
}
```

BINARY SEARCH: ANALYSIS

The input size is the item-size of the array: the longer the array the more work the algorithm has to do to search for the input key.
The basic operation is the comparison: input key vs. array item (three-way comparison).
The basic operation count $C_2(n)$ (version 2, recursive top-down) depends not only on the input size but also on the input content (array and key). So, we need best- and worst- cases.

This is the recursive definition of Binary Search algorithm (version 2, recursive top-down):

$$\mathrm{bs2}(a,k,l,r) = \begin{cases} -1, & \text{if l} > r. \\ \left\lfloor \frac{l+r}{2} \right\rfloor, & \text{if l} \le \text{r and } k = a[\left\lfloor \frac{l+r}{2} \right\rfloor]. \\ \mathrm{bs2}\left(a,k,l,\left\lfloor \frac{l+r}{2} \right\rfloor - 1\right), & \text{if l} \le \text{r and } k < a[\left\lfloor \frac{l+r}{2} \right\rfloor]. \\ \mathrm{bs2}\left(a,k,\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right), & \text{if l} \le \text{r and } k > a[\left\lfloor \frac{l+r}{2} \right\rfloor]. \end{cases}$$

BINARY SEARCH: ANALYSIS 2

- The **worst-case** inputs include all arrays that do not contain a given search key, as well as some successful searches. So, in these scenarios, the basic operation count $C_{2\text{-worst}}(n)$ is:

$$C_{2-\text{worst}}(n) = \begin{cases} 0, & \text{if } n = 0 \text{ (or } l > r\text{).} \\ 1, & \text{if } n = 1 \text{ (or } l = r\text{).} \\ C_{2-\text{worst}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1, & \text{if } n > 1 \text{ (or } l > r\text{).} \end{cases}$$

Note: To facilitate its conversion into a closed-form expression, the basic operation count has been defined using 1 input parameter only (n) representing the item-size of the search partition (from left to right indices, inclusive), or: n=right-left+1.

Note: This recursive definition includes only 1 recurrence (representing search partitions with odd and even item-sizes) because, in the worst-case, the Binary Search algorithm always selects the longer half in splitting the input search partition.

BINARY SEARCH: ANALYSIS 3

- The recursive definition of basic operation count $C_{2\text{-worst}}(n)$ for the **worst-case** can be converted into a closed-form expression by using the Backward Substitutions and the Smoothness Rule.

  Here we apply the Smoothness Rule with: $n = 2^k$ and $k = \log_2 n$

$$\begin{aligned} & C_{2-\text{worst}}(n) = C_{2-\text{worst}}(n/2) + 1 = \\ \text{subs} \quad & = [\, C_{2-\text{worst}}((n/2)/2) + 1 \,] + 1 = C_{2-\text{worst}}(n/4) + 2 = \\ & = [\, C_{2-\text{worst}}((n/4)/2) + 1 \,] + 2 = C_{2-\text{worst}}(n/8) + 3 \end{aligned}$$

$$\Downarrow$$

$$\text{pattern} \quad C_{2-\text{worst}}(n) = C_{2-\text{worst}}(2^k) = C_{2-\text{worst}}(n/2^i) + i, \quad \text{with } i \in [0, k]$$

$$\Downarrow$$

$$\text{closed} - \text{form} \quad C_{2-\text{worst}}(n/2^i) + i = C_{2-\text{worst}}(2^k/2^k) + k = 1 + \log_2 n$$

BINARY SEARCH: ANALYSIS 4

Now that we have the result of the worst-case analysis, we can quickly summarize the time efficiency analysis of the Binary Search algorithm (version 2, recursive top-down).

- The results of the **worst-case** analysis are:

$$C_{2-\text{worst}}(n) = 1 + \log_2 n \in \Theta(\log n)$$

- The results of the **best-case** analysis (successful searches immediate) are:

$$C_{2-\text{best}}(n) = 1 \in \Theta(1)$$

- So, the general basic operation count C(n) is in:

$$C_2(n) \in O(\log n) \quad \text{and} \quad C_2(n) \in \Omega(1)$$

VARIABLE-SIZE-DECREASE

In the variable-size-decrease design, the size of an instance is reduced by a variable amount on each iteration of the algorithm (e.g., from P(n) to P(n-1) or P(n-2)).
Usually, the amount of reduction in size depends on the specific values of the inputs.

Example: Euclid's algorithm for computing the GCD is a variable-size-decrease algorithm.
In fact, this algorithm is based on the formula: GCD(m,n) = GCD(n,m mod n)
In this formula, it is easy to demonstrate that the value of the 2nd argument is always smaller on the right-hand side (m mod n) than on the left-hand side (n).
However, the 2nd argument decreasing is neither driven by a constant nor by a constant factor.

SELECTION PROBLEM

Problem: The Selection Problem is the problem of finding the k^{th} smallest item in an array of n items. This item is called the k^{th} order statistic.

Of course, for k=1 or k=n, we can simply scan the array in question to find the smallest (min) or the largest (max) element, respectively.

A special version of the selection problem (finding the k^{th} smallest item in an array of n items) happens when $k=\lfloor n/2 \rfloor$. In this case, we have to find an item that is not greater than one half of the array items, and not less than the other half.
This middle value is called the median.

Sorting the array can help solving the problem, but there are more efficient ways. For example: using the partitioning of the array around an item p (see Figure 6).

Figure 6. Array partitioning around item p: all array items are rearranged so that the left part contains all items ⩽p, followed by the pivot p itself, followed by all items ⩾p.

QUICK SELECT: ALGORITHM

To solve the Selection Problem we can exploit any partitioning algorithm.

Quick Select (Using Variable-Size-Decrease)
Assume that s is the split index of the partition (i.e., index of the pivot p once the partitioning is completed), then:

- **If s=k-1:** then pivot p is the k^{th} smallest item (Selection Problem solved).
- **If s>k-1:** then the k^{th} smallest item of the input array can be found as the k^{th} smallest item in the left part (segment <p) of the partitioned array.
- **If s<k-1:** then the k^{th} smallest item of the input array can be found as the $(k-s)^{th}$ smallest item in the right part (segment ≥p) of the partitioned array.

So, if we do not find the k^{th} smallest item at index s, we reduce the problem size after the first partitioning. This smaller problem can still be solved by the same approach (i.e., recursively).

QUICK SELECT: IMPLEMENTATION

This is a Java implementation of the Quick Select by Variable-Size-Decrease algorithm (recursive, top-down).

```
public static int QuickSelect( int[] A, int left, int right, int k ) {
   // Note: any partitioning algorithm can be used here.
   int s = LomutoPartitioning( A, left, right );
   // Check pivot position after partitioning is completed.
   if( s == left+k-1 ) {
      // Current pivot is k-th smallest item, return pivot.
      return A[s];
   }
   else if( s > (left+k-1) ) {
      // k-th smallest item smaller than pivot...
      // ...call Quick Select on left segment (<pivot) with same k.
      return QuickSelect( A, left, s-1, k ); // Recursive call.
   }
   else {
      // k-th smallest item greater than pivot...
      // ...call Quick Select on right segment (>=pivot) with updated k (k-1-s).
      return QuickSelect( A, s+1, right, k-1-s ); // Recursive call.
   }
}
```

QUICK SELECT: ANALYSIS

The input size is the item-size (n) of the input subarray (from left to right, inclusive), that is: n=right-left+1. The magnitude of parameter k does not matter in determining the work done.

The basic operation is the comparison (of array indices) for the Quick Select, but we have also to consider the work done by the partitioning algorithm (e.g., Lomuto Partitioning).

The basic operation count C(n) depends not only on the input size but also on the input content (subarray and parameter k). So, we need best- and worst- cases.

- In the **best-case** the initial partitioning (using Lomuto Partitioning algorithm) immediately solves the Selection Problem. The Lomuto Partitioning algorithm requires n-1 comparisons, then we compare array indices once before returning, so:

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

QUICK SELECT: ANALYSIS 2

- In the **worst-case** the initial partitioning produces a very unbalanced partition (e.g., segment <p empty etc.). This extremely unbalanced split can happen on each of the partitionings required to solve the Selection Problem. If so, we will need n-1 partitionings in total (e.g., when k=n and the input array is sorted increasingly). Remember that each one of these partitionings processes a different subarray (smaller and smaller). So:

$$C_{worst}(n) = \underset{\text{Lomuto } 1}{(n-1)} + \cdots + \underset{\text{Lomuto n}-1}{1} = \sum_{i=n-1}^{1} i = \frac{n\,(n-1)}{2} \in \Theta(n^2)$$

- So, the general count C(n) is in:

$$C(n) \in O(n^2) \quad \text{and} \quad C(n) \in \Omega(n)$$

INTERPOLATION SEARCH: ALGORITHM

Problem: Given an input item, search it in a sorted array, returning its position (if found).

Interpolation Search (Using Variable-Size-Decrease)
Unlike Binary Search (which compares search key with middle value in given sorted array), Interpolation Search considers the search key value to find the element to be compared with.

Interpolation Search mimics the way we search for a name in a telephone book (i.e., if we search for "Brown", we open the telephone book not in the middle but very close to the beginning etc.).

INTERPOLATION SEARCH: ALGORITHM 2

In processing the subarray from A[left] to A[right], the Interpolation Search algorithm assumes that the array values increase linearly (uniformly distributed) (see Figure 7).

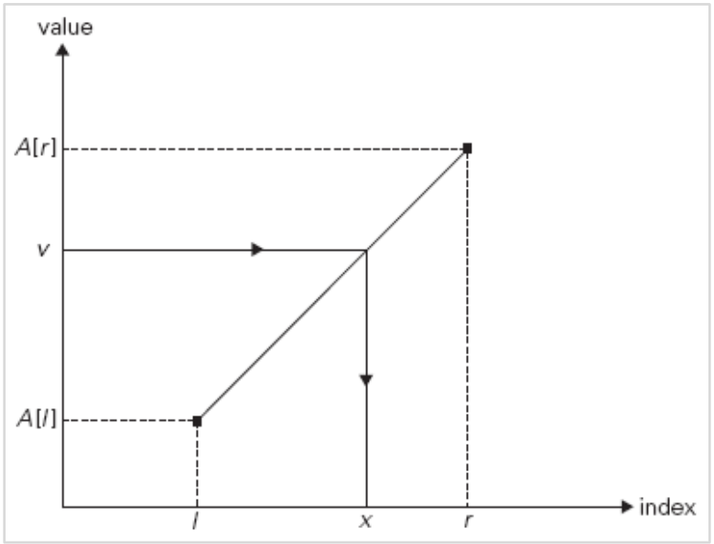

Figure 7. Probing index generation in Interpolation Search: the search key (v) is compared with subarray first/last items to approximate its location (x).

$$y - y_1 = \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1)$$

linear equation between 2 2D points

$$i = left + \left\lfloor \frac{(k - \mathrm{A}[left]) \times (right - left)}{\mathrm{A}[right] - \mathrm{A}[left]} \right\rfloor$$

interpolation search probing index

INTERPOLATION SEARCH: IMPLEMENTATION

Java implementation of Interpolation Search (Variable-Size-Decrease, recursive, top-down).

```
public static int InterpolationSearch( int[] A, int left, int right, int k ) {
   // BASE CASES: check if search partition is empty or trivial (1 item only).
   if( left > right ) { return -1; }
   else if( left == right ) {
      if( A[left] == k ) { return left; }
      else { return -1; }
   }
   else {
      // Input array is sorted, so search key has to be in first/last items range.
      if( ( k >= A[left] ) && ( k <= A[right] ) ) {
         // Generate probing index for search key assuming uniform distribution.
         int i = left + (((right-left) / (A[right]-A[left])) * (k-A[left]));
         // Check if search key is at probing index.
         if( A[i] == k ) { return i; } // Search key found.
         // If probing item less than search key, search in right subarray.
         else if( A[i] < k ) { return InterpolationSearch(A, i+1, right, k); }
         // If probing item greater than search key, search in left subarray.
         else { return InterpolationSearch( A, left, i-1, k ); }
      }
      else { return -1; } // Input search key out of range, search failed.
   }
}
```

INTERPOLATION SEARCH: ANALYSIS

The input size is the item-size of the array.

The basic operation is the comparison: input key vs. array item (three-way comparison).

The basic operation count C(n) depends not only on the input size but also on the input content (array and key). So, we need best- and worst- cases.

- In **worst-case**, search range shrinks by 1 at each step: $C_{worst}(n) = n \in \Theta(n)$
- In **best-case**, the first probe finds the search key: $C_{best}(n) = 1 \in \Theta(1)$
- So, the general count C(n) is in: $C(n) \in O(n)$ and $C(n) \in \Omega(1)$

Note: Interpolation Search is preferable to Binary Search only for very large arrays and/or in cases when comparisons or array accesses are particularly expensive.