PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Brute Force: Selection Sort

Dr. Shylaja S S

Brute Force: Brute Force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

The "force" implied by the strategy's definition is that of a computer and not that of one's intellect. "Just do it!" would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute a^n for a given number **a** and a nonnegative integer **n**. Though this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design techniques, including the brute-force approach. (Also note that computing a^n mod m for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation, a^n = a*…* a. (n times)

This suggests simply computing a^n by multiplying **1** by **a** n times.

*In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of:

- systematically enumerating all possible candidates for the solution
- checking whether each candidate satisfies the problem's statement

A brute-force algorithm to find the divisors of a natural number n would

- enumerate all integers from 1 to n
- check whether each of them divides n without remainder

A brute-force approach for the eight queens puzzle would

- examine all possible arrangements of 8 pieces on the 64-square chessboard
- check whether each (queen) piece can attack any other, for each arrangement

The brute-force method for finding an item in a table (linear search) checks all entries of the table, sequentially, with the item.

*https://en.wikipedia.org/wiki/Brute-force_search

A brute-force search is simple to implement, and will always find a solution if it exists. But, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (Combinatorial explosion)

Brute-force search is typically used:

- when the problem size is limited
- when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size
- when the simplicity of implementation is more important than speed

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n - 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the i^{th} pass through the list, which we number from 0 to n - 2, the algorithm searches for the smallest item among the last n-i elements and swaps it with A_i.

$A[0] \leq A[1] \leq A[2] \ldots \leq A[i-1] \mid A[i], \ldots\ldots, A[min], \ldots., A[n-1]$

in their final positions the last n – i elements

After n - 1 passes, the list is sorted.

Here is a pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array.

```
ALGORITHM  SelectionSort(A[0 .. n -1])
//Sorts a given array by selection sort
//Input: An array A[0 .. n - 1]  of orderable elements
//Output: Array A[0 .. n - 1]  sorted in ascending order
for i <- 0 to n - 2 do
        min <- i
        for j <- i+1 to n-1 do
                if A[j] <  A[min]  min  <- j
        swap  A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Fig. 1.

| 89 45 68 90 29 34 **17**
17 | 45 68 90 **29** 34 89
17 29 | 68 90 45 **34** 89
17 29 34 | 90 **45** 68 89
17 29 34 45 | 90 **68** 89
17 29 34 45 68 | 90 **89**
17 29 34 45 68 89 | 90

Fig. 1: Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

The analysis of selection sort is straightforward. The input's size is given by the number of elements n; the algorithm's basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array's size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Selection Sort is a $\Theta(n^2)$ algorithm on all inputs.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Bubble Sort

Dr. Shylaja S S

Bubble Sort

Bubble Sort is a brute-force application to the sorting problem. In bubble sort, we compare the adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on until, after n - 1 passes, the list is sorted. Pass i ($0 \leq i \leq n-2$) of bubble sort can be represented by the following diagram:

$A[0], A[1], A[2], \ldots, A[j] \overset{?}{\leftrightarrow} A[j+1], \ldots, A[n-i-1] \mid A[n-i] \leq \ldots. \leq A[n-1]$

in their final positions

Here is a pseudocode of this algorithm.

```
ALGORITHM BubbleSort(A[0 .. n - 1])
//Sorts a given array by bubble sort in their final positions
//Input: An array A[0 .. n - 1] of orderable elements
//Output: Array A[0 .. n- 1] sorted in ascending order
for i <-- 0 to n - 2 do
    for j <-- 0 to n - 2 - i do
        if A[ j + 1 ] < A[ j ] swap A[ j ] and A[ j + 1 ]
```

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Fig. 1.

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
45 68 89 29 90 ? 34 17
45 68 89 29 34 90 ? 17
45 68 89 29 34 17 | 90

45 ? 68 89 29 34 17 | 90
45 68 ? 89 29 34 17 | 90
45 68 89 ? 29 34 17 | 90
45 68 29 89 ? 34 17 | 90
45 68 29 34 89 ? 17 | 90
45 68 29 34 17 | 89 90

Fig. 1: First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

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

Bubble Sort is a $\Theta(n^2)$ algorithm.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Sequential Search

Dr. Shylaja S S

Sequential Search

The sequential search algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate a check for the list's end on each iteration of the algorithm. Here is a pseudocode for this enhanced version, with its input implemented as an array.

```
AlGORITHM  SequentialSearch2(A[0 .. n ],  K)
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0 .. n -1] whose value is
// equal to K or -1 if no such element is found
A[n]<---K
i<---0
while A[i] ≠K  do
        i<--- i + 1
if i  <  n return i
else return -1
```

Sequential Search is a Θ(n) algorithm.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

For key = 33, 6 is returned
For key = 50, -1 is returned (50 is stored in position 10 in the array)

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

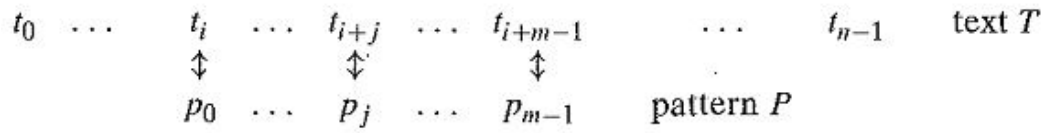
String Matching

Dr. Shylaja S S

String Matching

Given a string of n characters called the **text** and a string of m characters (m ≤n) called the **pattern**; find a substring of the text that matches the pattern. To put it more precisely, we want to find i - the index of the leftmost character of the first matching substring in the text-such that

$t_i = p_0 , \ldots , t_{i+j} = p_j , \ldots . , t_{i+m-1} = p_{m-1}$:



If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text which can still be a beginning of a matching substring is n - m (provided the text's positions are indexed from 0 to n- 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
ALGORITHM BruteForceStringMatch(T[0 .. n -1], P[0 .. m -1])
//Implements brute-force string matching
//Input: An array T[0 .. n - 1] of n characters representing a text and an array
//P[0 .. m - 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a matching
//substring or -1 if the search is unsuccessful
for i ← 0 to n-m do
        j ←0
        while j < m and P[j] = T[i + j] do
                j ← j+1
        if j = m return i
return -1
```

Example:

N 0 B 0 D Y _ N 0 T I C E D _ H I M

N 0 T

  N 0 T

    N 0 T

      N 0 T

        N 0 T

          N 0 T

            N 0 T

              N 0 T

Fig 1: Example of brute-force string matching. (The pattern's characters that are compared with their text counterparts are in bold type.)

Note that for this example, the algorithm shifts the pattern almost always after a single character comparison. However, the worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n-m+1 tries. Thus, in the worst case, the algorithm is in Θ(nm). For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again). Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e., Θ(n + m) = Θ(n).

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

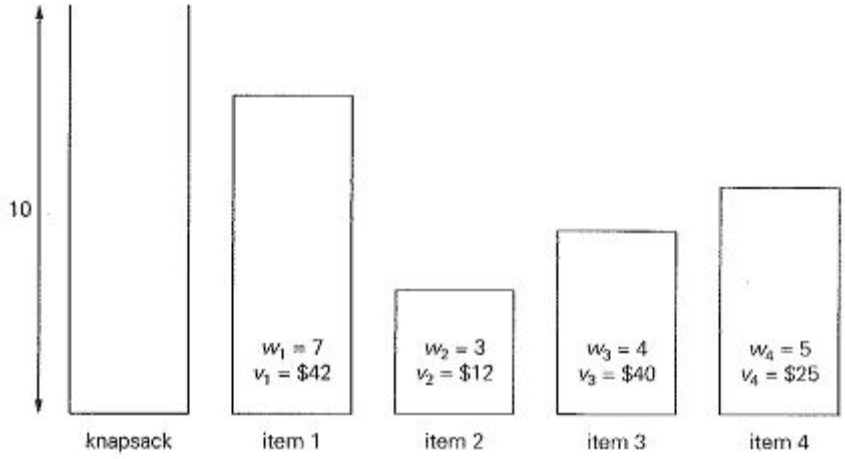
UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Exhaustive Search: Knapsack Problem

Dr. Shylaja S S

Knapsack Problem

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. If you do not like the idea of putting yourself in the shoes of a thief who wants to steal the most valuable loot that fits into his knapsack, think about a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity. Fig. 1 presents a small instance of the knapsack problem.



(a)

Subset	Total weight	Total value
∅	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$36
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

Fig. 1: (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. (The information about the optimal selection is in bold.)

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack's capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Fig. 1a is given in Fig. 1b. Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm no matter how efficiently individual subsets are generated.

Thus, for both the traveling salesman and knapsack problems, exhaustive search leads to algorithms that are extremely inefficient on every input. In fact, these two problems are the best-known examples of so-called NP-hard problems. No polynomial-time algorithm is known for any NP-hard problem. Moreover, most computer scientists believe that such algorithms do not exist, although this very important conjecture has never been proven. More sophisticated approaches - backtracking and branch-and-bound enable us to solve some but not all instances of these (and similar) problems in less than exponential time. Alternatively, we can use one of many approximation algorithms.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Exhaustive Search: Assignment Problem

Dr. Shylaja S S

Assignment Problem

There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the i^{th} person is assigned to the j^{th} job is a known quantity C[i, j] for each pair i, j = 1, 2, ... , n. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs C[i, j]:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

It is easy to see that an instance of the assignment problem is completely specified by its cost matrix C. In terms of this matrix, the problem calls for a selection of one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible. Note that no obvious strategy for finding a solution works here. For example, we cannot select the smallest element in each row because the smallest elements may happen to be in the same column. In fact, the smallest element in the entire matrix need not be a component of an optimal solution. Thus, opting for the exhaustive search may appear as an unavoidable evil.

We can describe feasible solutions to the assignment problem as n-tuples $<j_1, \ldots, j_n>$ in which the i^{th} component, i = 1, ... , n, indicates the column of the element selected in the i^{th} row (i.e., the job number assigned to the i^{th} person). For example, for the cost matrix above, <2, 3, 4, 1> indicates a feasible assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1. The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and

permutations of the first n integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers 1, 2, ..., n, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are shown in Fig. 1; you may complete the remaining.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23
etc.

Fig. 1: First few iterations of solving a small instance of the assignment problem by exhaustive search

Since the number of permutations to be considered for the general case of the assignment problem is n!, exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the Hungarian method after the Hungarian mathematicians Konig and Egervary whose work underlies the method.

PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS251: Design and Analysis of Algorithms (4-0-0-4-4)

Exhaustive Search: Travelling Salesman Problem

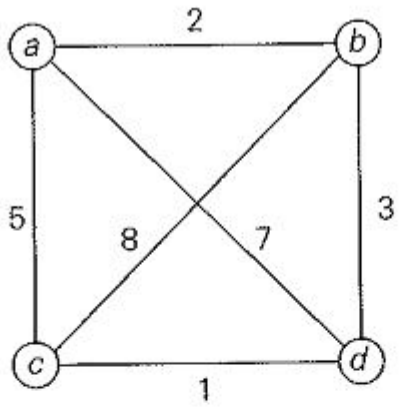
Dr. Shylaja S S

Many important problems require finding an element with a special property in a domain that grows exponentially (or faster) with an instance size. Typically, such problems arise in situations that involve-explicitly or implicitly-combinatorial objects such as permutations, combinations, and subsets of a given set. Many such problems are optimization problems: they ask to find an element that maximizes or minimizes some desired characteristic such as a path's length or an assignment's cost.

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem's domain, selecting those of them that satisfy all the constraints, and then finding a desired element (e.g., the one that optimizes some objective function). Note that though the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects. We assume here that they exist. We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem. In this section, we shall discuss the traveling salesman problem.

The traveling salesman problem (TSP) has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems. In layman's terms, the problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started. The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805-1865), who became interested in such cycles as an application of his algebraic discoveries.) It is easy to see that a Hamiltonian circuit can be also defined as a sequence of $n + 1$ adjacent vertices $v_{i0}, v_{i1}, \ldots, V_{in-1}, v_{i0}$, where the first vertex of the sequence is the same as the last one while all the other $n - 1$ vertices are distinct. Further, we can assume, with no loss of generality, that all circuits start and end at one particular vertex (they are cycles after all, are they

not?). Thus, we can get all the tours by generating all the permutations of n - 1 intermediate cities, compute the tour lengths, and find the shortest among them. Fig. 1 presents a small instance of the problem and its solution by this method.



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Fig. 1: Solution to a small instance of the traveling salesman problem by exhaustive search

An inspection of Fig. 1 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half. We could, for example, choose any two intermediate vertices, say, B and C, and then consider only permutations in which B precedes C. (This trick implicitly defines a tour's direction.)

This improvement cannot brighten the efficiency picture much, however. The

total number of permutations needed will still be (n- 1)!/2, which makes the exhaustive-search approach impractical for all but very small values of n. On the other hand, if you always see your glass as half-full, you can claim that cutting the work by half is nothing to sneeze at, even if you solve a small instance of the problem, especially by hand. Also note that had we not limited our investigation to the circuits starting at the same vertex, the number of permutations would have been even larger, by a factor of n.

The Exhaustive Search solution to the Travelling Salesman problem can be obtained by keeping the origin city constant and generating permutations of all the other n – 1 cities. Thus, the total number of permutations needed will be (n – 1)!

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 its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up. The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.
There are three major variations of decrease-and-conquer:
decrease by a constant
decrease by a constant factor
variable size decrease

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.
Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size $n-1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition or "bottom up" by multiplying 1 by a n times. (Yes, it is the same as the brute-force algorithm, but we have come to it by a different thought process.)

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

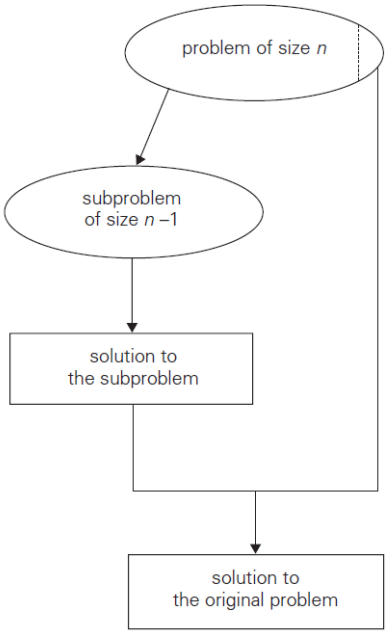


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. For an example, let us revisit the exponentiation problem. If the instance of
size n is to compute a^n, the instance of half its size is to compute $an/2$, with the obvious relationship between the two: $an = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

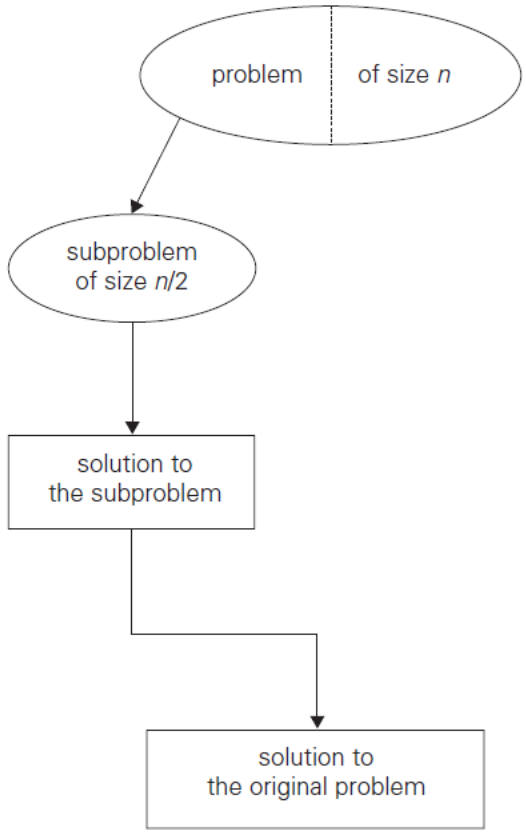


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Finally, in the ***variable-size-decrease*** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid' s algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula gcd*(m, n)* = gcd*(n, m* mod *n)*.

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

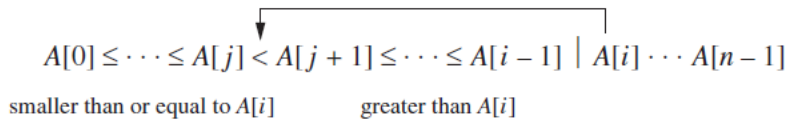


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

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 4.4 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.

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

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort (see Section 3.1).

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

Topological Sorting

directed graph, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 4.5a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists. Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be
more complex than for undirected graphs.
Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, *a, b, a* is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back
edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***. For topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary
but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution

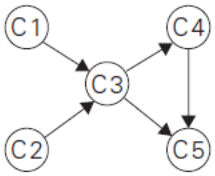


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

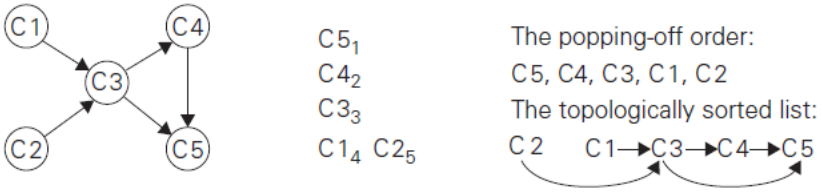


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem. The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8. Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

Generating Permutations and Subsets

The most important types of combinatorial objects are permutations, combinations, and subsets of a given set. There are n! permutations of {1, 2, . . . , n}. It is important in many instances to generate a list of such permutations.

Generating Permutations

We are given a sequence of numbers from 1 to n. Each permutation in the sequence that we need to generate should differ from the previous permutation by swapping just two adjacent elements of the sequence.

Input : n = 3

Output : 123 132 312 321 231 213

Input : n = 4

Output : 1234 1243 1423 4123 4132

1432 1342 1324 3124 3142 3412 4312

4321 3421 3241 3214 2314 2341 2431

4231 4213 2413 2143 2134

The Johnson and Trotter algorithm doesn't require to store all permutations of size n-1 and doesn't require going through all shorter permutations. Instead, it keeps track of the direction of each element of the permutation.

1. Find out the largest mobile integer in a particular sequence. **A directed integer is said to be mobile if it is greater than its immediate neighbor in the direction it is looking at**.
2. Switch this mobile integer and the adjacent integer to which its direction points.
3. Switch the direction of all the elements whose value is greater than the mobile integer value.
4. Repeat the step 1 until unless there is no mobile integer left in the sequence.

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

Here is an application of this algorithm for $n = 3$, with the largest mobile element shown in bold:

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

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
  let i be its largest index such that $a_i < a_{i+1}$ $\quad //a_{i+1} > a_{i+2} > \cdots > a_n$
  find the largest index j such that $a_i < a_j$ $\quad //j \geq i+1$ since $a_i < a_{i+1}$
  swap a_i with a_j $\quad //a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
  reverse the order of the elements from a_{i+1} to a_n inclusive
  add the new permutation to the list

Generating Subsets

knapsack problem, which asks to find the most valuable subset of items that fits a knapsack of a given capacity. The exhaustive-search approach to solving this problem discussed there was based on generating all subsets of a given set of items. In this section, we discuss algorithms
for generating all 2^n subsets of an abstract set A

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

FIGURE 4.10 Generating subsets bottom up.

A more challenging question is whether 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. (In the language of subsets, we want every subset to differ from its immediate predecessor by either an addition or a deletion, but not both, of a single element.) The answer to this question is yes. For example, for $n = 3$, we can get

$$000 \quad 001 \quad 011 \quad 010 \quad 110 \quad 111 \quad 101 \quad 100.$$

Such a sequence of bit strings is called Binary reflected Gray code.

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