Analysis and Design of Algorithms

Chapter 3: Brute Force

School of Software Engineering © Yanling Xu

Brute Force

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Just do it

- ***Example:***
  - *Computing a^n ($a > 0$, n a nonnegative integer)*
  - *Computing $n!$*
  - *Multiplying two matrices*
  - *Searching for a key of a given value in a list*
  - *Consecutive Integer Algorithm for gcd (m,n)*

Brute-Force Sorting Alg. — Selection Sort

Idea of Selection Sort

- *Problem*

  Given an array of n orderable items (e.g. numbers, characters from some alphabet, character strings), rearrange them in non-decreasing order

Selection Sort

- *Idea*
  - Scan the entire array to find its smallest element and swap it with the first element. — put the smallest element in its final position in the sorted array
  - starting with the second element, to find the smallest among the next n-1 elements and swap it with the second element. — put the second smallest element in its final position in the sorted array
  - Generally, on pass i ($0 \le i \le n-2$), find the smallest element in $A[i..n-1]$ and swap it with $A[i]$:
  - After n-1 passes, the array is sorted

$$A[0] \le \; . \;\; . \;\; . \; \le A[i-1] \;\; | \;\; A[i], \; . \;\; . \;\; . \; , A[min], . \;\; . \; ., A[n-1]$$

in their final positions

Selection Sort

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 $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
      if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

Selection Sort

- ***Example:***

Selection Sort on the list {89, 45, 68, 90, 29, 34, 17 }

| 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

FIGURE 3.1 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.

Selection Sort

Analysis of Selection Sort

- ***Basic operation: key comparison $A[j] < A[min]$***
- ***Input size: number of elements, n***
- ***Time efficiency*** $\Theta(n^2)$

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

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

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

- *number of key swaps:* $\Theta(n)$

Brute-Force Sorting Alg. — Bubble Sort

Idea of Bubble Sort

Idea

- Compare adjacent elements of the list and exchange them if they are out of order
- By doing it repeatedly, we end up “bubbling” the largest element to the last position on the list
- The next past bubbles up the second largest element, and so on until, after *n*-1 passes, the list is sorted
- Pass *i*

$$A_0 \ldots\ldots A_j \overset{?}{\longleftrightarrow} A_{j+1} \ldots\ldots A_{n-i-1} \mid \underbrace{A_{n-i} \leq \ldots \leq A_{n-1}}_{\text{in their final positons}}$$

Bubble Sort

ALGORITHM *BubbleSort* (*A* [0…*n*-1])

{

  // Sorts a given array by bubble sort;

  // Input: An array *A*[0…*n*-1] of orderable elements

  // Output: Array *A*[0…*n*-1] sorted in ascending order

  For $i \leftarrow 0$ to n-2 do

    For $j \leftarrow 0$ to n-2-i do

      if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

}

Bubble Sort

- ***Example:***

Bubble Sort on the list {89, 45, 68, 90, 29, 34, 17 }

89	$\overset{?}{\longleftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\longleftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\longleftrightarrow}$	90	$\overset{?}{\longleftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\longleftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\longleftrightarrow}$	17
45		68		89		29		34		17	$\vert$	90
45	$\overset{?}{\longleftrightarrow}$	68	$\overset{?}{\longleftrightarrow}$	89	$\overset{?}{\longleftrightarrow}$	29		34		17	$\vert$	90
45		68		29		89	$\overset{?}{\longleftrightarrow}$	34		17	$\vert$	90
45		68		29		34		89	$\overset{?}{\longleftrightarrow}$	17	$\vert$	90
45		68		29		34		17	$\vert$	89		

Bubble Sort

Analysis of Bubble Sort

- ***Basic operation: key comparison***
- ***Input size: number of elements, n***
- ***Time efficiency*** $\boldsymbol{\Theta(n^2)}$

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

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

- *number of key swaps: depends on the input*

$$S_{worst}(n) = C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Thinking: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm

Brute-Force String Matching

Idea of Brute-Force String Matching

text: a (longer) string of *n* characters to search in

pattern: a string of *m* characters to search for (m <= n)

- *Problem*

find a substring in the text that matches the pattern,

precisely, find *i* — the index of the leftmost character of the first matching substring in the text — such that

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

Brute-Force String Matching

- *Idea*
  - S1: Align pattern against the first *m* characters of the text
  - S2: compare corresponding pairs of characters from left to right, starting with the first character of the pattern and its counter part in the text, until

    _ Case1: all *m* pairs are found to match (successful search); or

    _ Case2: a mismatching pair is detected
  - S3: In Case2, the text is not yet exhausted, realign pattern one position to the right and repeat S2, starting again with the first left pair.

  Note: the last position in the text which can still be a beginning of a matching substring is ***n-m***

Brute-Force String Matching

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
```

Brute-Force String Matching

- ***Example:***

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

Brute-Force String Matching

Analysis of Brute-Force String Matching

- ***Basic operation: key comparison***
- ***Input size: n, m***
- ***Time efficiency***
  - *worst case:* it has to make all m comparisons before shifting the pattern, and this can happen for each of the n-m+1 tries.

    $C_{worst} = \Theta(nm)$

  - *average case:* for a typical word search, we can expect most shifts would happen after very few comparisons

    $C_{avg} = \Theta(n+m) = \Theta(n)$

Closest-Pair Problem

Idea of Closest-Pair Problem

- *Problem*

  Find the two closest points in a set of n points (in the two-dimensional Cartesian plane).

- *Idea*

  Compute the Euclidean distance between every pair of distinct points;

  and return the indexes of the points for which the distance is the smallest.

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Closest-Pair Problem

ALGORITHM *BruteForceClosestPoints*(P)

//Input: A list P of n ($n \geq 2$) points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$
//Output: Indices $index1$ and $index2$ of the closest pair of points
$dmin \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$ //$sqrt$ is the square root function
        if $d < dmin$
            $dmin \leftarrow d$; $index1 \leftarrow i$; $index2 \leftarrow j$
return $index1, index2$

Closest-Pair Problem

Idea of Closest-Pair Problem

- *How to make it faster?*

  The basic operation of the algorithm is computing the Euclidean distance between two points.

  The square root is a complex operation who’s result is often irrational, therefore the results can be found only approximately. Computing such operations are not trivial.

  —→ One can *avoid* computing square roots by comparing distance squares instead.

Closest-Pair Problem

Analysis of Closest-Pair Problem

- *Basic operation: squaring a number*
- *Input size: number of points, n*
- *Time efficiency $\Theta(n^2)$*

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

Brute-Force Polynomial Evaluation

Idea of Polynomial Evaluation

- *Problem*

  Find the value of polynomial

  $$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

  at a point $x = x_0$

- *Idea*

Brute-Force Polynomial Evaluation

$p \leftarrow 0.0$

for $i \leftarrow n$ **down to** 0 **do**

    $power \leftarrow 1$

        for $j \leftarrow 1$ **to** i **do**   **//compute x^i**

            $power \leftarrow power * x$

        $p \leftarrow p + a[i] * power$

    return p

Brute-Force Polynomial Evaluation

Better Polynomial Evaluation

evaluating from right to left:

$p \leftarrow a[0]$

$power \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**

$\quad\quad power \leftarrow power * x$

$\quad p \leftarrow p + a[i] * power$

return p

Exhaustive Search

Problem

searching for an element with a special property, in a domain that grows exponentially (or faster) with an instance size,

usually involve combinatorial objects such as permutations, combinations, or subsets of a set.

Many such problems are optimization problems, to find an element that maximizes or minimizes some desired characteristic

such as a path's length or an assignment's cost

Exhaustive Search

Exhaustive Search— Brute-Force for combinatorial

- generate a list of all potential solutions to the problem in a systematic manner
- selecting those of them that satisfy all the constraints
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- then search ends, announce the desired solution(s) found (e.g. the one that optimizes some objective function)
- *typically requires for generating certain combinatorial objects*

Exhaustive Search: *Traveling Salesman Problem*

Idea

- *Problem*

  Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city

- *Idea*

  - weighted graph:

    vertices: cities

    edge weights: distances

  - Alternatively: To find shortest Hamiltonian circuit in a weighted connected graph

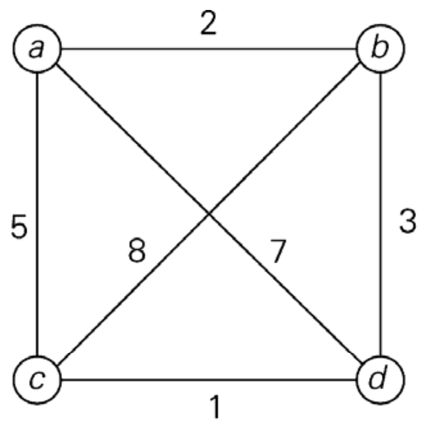
  Hamiltonian circuit: a cycle that passes through all the vertices of the graph exactly once

Exhaustive Search: Traveling Salesman Problem

- *Idea*
  - Hamiltonian circuit can be defined as a sequence of n+1 adjacent vertices $v_{i0}, v_{i1}, v_{i2}, \ldots\ldots, v_{in-1}, v_{i0}$
  - generating all the permutations of n-1 intermediate cities
  - computing the tour lengths
  - find the shortest among them

Traveling Salesman Problem

- ***Example:***



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

Traveling Salesman Problem

Analysis of Exhaustive Search for TSP

- *number of permutations* $(n\text{-}1)!$

Exhaustive Search: *Knapsack Problem*

- ***Idea***
  - *Problem*

Given

weights: $w_1 \quad w_2 \; \ldots \; w_n$
values: $v_1 \quad v_2 \; \ldots \; v_n$
a knapsack of capacity W

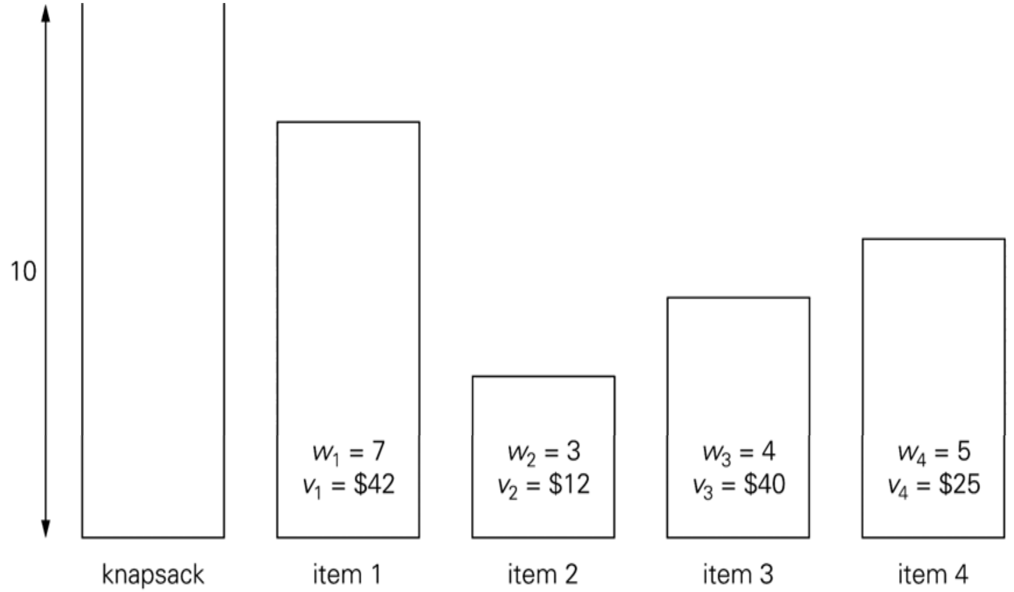
find the most valuable subset of the items that fit into the knapsack

Exhaustive Search: Knapsack Problem

- *Idea*
  - generating all subsets of the set of *n* items given
  - computing the total weight of each feasible subset (i.e. the ones with the total weight not exceeding the knapsack's capacity)
  - finding a subset of the largest value among them

Knapsack Problem

- ***Example:***



(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)

Knapsack Problem

Analysis of Exhaustive Search for Knapsack

- *number of subsets for an n-element set* 2^n

For Exhaustive Search for Knapsack Problem and TSP problem,

- *examples of so-called NP-hard problem*
- *no polynomial-time algorithm is known for NP-hard problem*

Exhaustive Search: *Assignment Problem*

Idea

Problem

There are *n* people who need to be assigned to *n* jobs, one person per job.

each person is assigned to exactly one job, and each job is assigned to exactly one person

The cost of assigning person *i* to job *j* is *C* [*i, j*]

Find an assignment that minimizes the total cost.

Exhaustive Search: **Assignment Problem**

- *Idea*

describe the feasible solutions to the Assignment Problem as n-tuples $<j_1, \ldots, j_n>$ *in which the i-th component indicates the column of the element selected in the i-th row (i.e. job number assigned to the i-th person)*

- generating all legitimate assignments,
- compute their costs
- select the cheapest one

Assignment Problem

- ***Example:***

	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

Pose the problem as the one about a cost matrix:

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

$\langle 1, 2, 3, 4 \rangle$	cost = 9 + 4 + 1 + 4 = 18
$\langle 1, 2, 4, 3 \rangle$	cost = 9 + 4 + 8 + 9 = 30
$\langle 1, 3, 2, 4 \rangle$	cost = 9 + 3 + 8 + 4 = 24
$\langle 1, 3, 4, 2 \rangle$	cost = 9 + 3 + 8 + 6 = 26
$\langle 1, 4, 2, 3 \rangle$	cost = 9 + 7 + 8 + 9 = 33
$\langle 1, 4, 3, 2 \rangle$	cost = 9 + 7 + 1 + 6 = 23

etc.

Assignment Problem

Analysis of Exhaustive Search for Assignment

- *number of permutations* $n!$

- *no known polynomial-time algorithms for problems whose domain grows exponentially with instance size*

Final Comments

- *Brute-Force Strengths and Weaknesses*
  - *Strengths*
    - *wide applicability*
    - *simplicity*
    - *yields reasonable algorithms for some important problems*
      (e.g., matrix multiplication, sorting, searching, string matching)
  - *Weaknesses*
    - *rarely yields efficient algorithms*
    - *some brute-force algorithms are unacceptably slow*
    - *not as constructive as some other design techniques*

Final Comments

- *Comments on Exhaustive Search*
  - *Brute-force is a straightforward approach to solving a problem, directly based on the definitions or statement of a problem*
  - *Exhaustive-search algorithms run in a realistic amount of time only on very small instances*
  - *In some cases, there are much better alternatives*
    - *Euler circuits*
    - *shortest paths*
    - *minimum spanning tree*
    - *assignment problem*
  - *In many cases, exhaustive search or its variation is the only known way to get exact solution*