Brute Force and Exhaustive Search

CMSC 142: Design and Analysis of Algorithms

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

Overview

Brute Force

Exhaustive Search

Brute-Force Strategy

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

Examples of brute-force algorithms

1. Consecutive checking algorithm for computing $\gcd(m, n)$
2. Computing $n!$
3. Definition-based algorithm for matrix multiplication
4. Sequential search

Brute-force Sorting Algorithms

Selection Sort

- First, scan the list of n elements (say in an array A) to find its smallest element and swap it with the first element.
- Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements.
- Generally, on the i^{th} pass where $(0 \leq i \leq n-2)$, find the smallest element in $A[i \ldots n-1]$ and swap it with $A[i]$.

$$\underbrace{A_0 \leq A_1 \leq \cdots \leq A_{i-1}}_{\text{in their final positions}} \;\Big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

- After $n-1$ passes, list is sorted.

Brute-force Sorting Algorithms

Pseudocode for the selection sort (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 nondecreasing 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]
```

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Brute-force Sorting Algorithms

Bubble Sort

- Compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, the largest element "bubbles up" to the last position on the list.
- The next pass bubbles up the second largest element, and so on. The i^{th} pass of the bubble sort where $0 \leq i \leq n-2$ is represented as:

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

- After $n-1$ passes, list is sorted.

Brute-force Sorting Algorithms

Pseudocode for bubble sort (list implemented as array):

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 nondecreasing 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 $A[j+1]$

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Brute-force Searching: Sequential Search

- The algorithm 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).

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//              or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Brute-force Searching: Modified Sequential Search

- Appending the search key to the end of the list will make search successful.

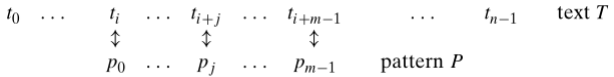
ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//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
```

Brute-Force String Matching

- Given a string of n characters called the **text** and a string of m characters ($m \leq n$) called the **pattern**, find a substring of the text that matches the pattern.
- In other words, find i, the index of the leftmost character of the first matching substring in the text such that

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



- A string-matching algorithm may continue working until the entire text is exhausted if all matches need to be found.

Brute-Force String Matching

Brute-force Algorithm

Step 1. Align pattern at beginning of text.

Step 2. Moving from left to right, compare each character of pattern to the corresponding character in text until

- all characters are found to match (successful search); or
- a mismatch is detected.

Step 3. While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2.

Brute-Force String Matching

Illustration. An operation of the previous algorithm is illustrated below.

```
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
```

Remark. The pattern's characters that are compared with their text counterparts are in bold type.

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
```

What is the efficiency class of the algorithm?

Brute-force Polynomial Evaluation

Problem. Given $p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$, find $p(x = x_0)$.

Brute-force Algorithm

//Input: An array $C[0 \ldots n]$ of coefficients of the polynomial $p(x)$; x_0
//Output: $p(x_0)$m function value of p at $x = x_0$
$p \leftarrow 0.0$, $x \leftarrow x_0$
for $i \leftarrow$ n **downto** 0 do
    power $\leftarrow 1$
        for $j \leftarrow 1$ **to** i do
            $power \leftarrow power * x$
        $p \leftarrow p + C[i] * power$
return p

Brute-force Polynomial Evaluation

Problem. Given $p(x) = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0$, find $p(x = x_0)$.

Brute-force Algorithm (Improved)

//Function evaluation from right to left
//Input: An array $C[0 \ldots n]$ of coefficients of the polynomial $p(x)$; x_0
//Output: $p(x_0)$m function value of p at $x = x_0$
$p \leftarrow$ C[0], $x \leftarrow x_0$
power $\leftarrow 1$
for $i \leftarrow 1$ **to** n do
    $power \leftarrow power * x$
    $p \leftarrow p + C[i] * power$
return p

Closest-Pair Problem by Brute Force

Closest-Pair Problem

- The closest-pair problem calls for finding two closest points in a set of n points.
- Assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is the standard Euclidean distance

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

- Brute-force algorithm: Compute the distance between every pair of distinct points and return the indices of points for which the distance is smallest.

Closest-Pair Problem by Brute Force

Pseudocode for the closest-pair problem by brute-force:

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force
//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$
//Output: The distance between the closest pair of points
$d \leftarrow \infty$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    for $j \leftarrow i + 1$ **to** n **do**
        $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root
return d

Determine the number of times the basic operation is executed and the efficiency class of this algorithm.

Brute-Force Strategy

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

Overview

Brute Force

Exhaustive Search

Exhaustive Search

Exhaustive search is a brute-force approach to combinatorial problems. A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method

- generate a list of all potential solutions to the problem in a systematic manner
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

Exhaustive Search

Traveling Salesman Problem (TSP)

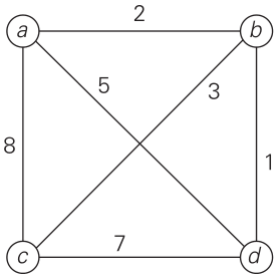
- The TSP asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
- Alternatively: Find the shortest **Hamiltonian circuit** in a weighted connected graph

A Hamiltonian circuit is a cycle that passes through all the vertices of the graph exactly once.

Exhaustive Search

Example

Consider the graph below. Find the shortest Hamiltonian circuit of the graph that starts with vertex a.



Exhaustive Search

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.

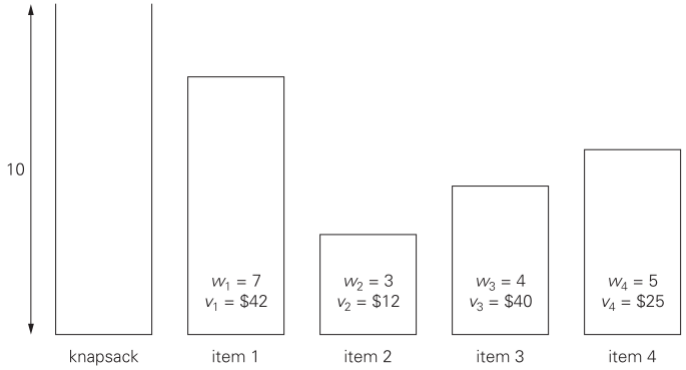
Algorithmic Plan

1. Generate all the subsets of the set of n items given.
2. Compute the total weight of each subset to identify *feasible* subsets.
3. Find a subset of the largest value among them.

Exhaustive Search

Example

Solve the knapsack problem below.



Exhaustive Search

Remarks

- For the Traveling Salesman Problem and Knapsack problem, exhaustive search leads to an algorithm that is inefficient on every input.
- The Traveling Salesman Problem and Knapsack problem are **NP-hard problems**.

  No polynomial-time algorithm is known for any NP-hard problem.

- Some but not all instances of NP-hard problems are solved by other more sophisticated approaches (*backtracking*, *branch and bound*)
- Alternatively, greedy algorithms or approximation algorithms can be used to "solve" the problem

Exhaustive Search

The Assignment Problem

- There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is $C[i, j]$. Find an assignment that minimizes the total cost.

Algorithmic Plan

Generate all legitimate assignments, compute their costs, and select the cheapest one.

Exhaustive Search

Example

Solve the assignment problem.

	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

Exhaustive Search Strategy

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances wide applicability (*and slow and computationally expensive, especially for problems with a large search space*)
- In some cases, there are much better alternatives!
- In many cases, exhaustive search or its variation is the only known way to get exact solution.

End of Lecture