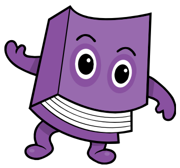
COT 6405
ANLYSIS OF ALGORITHMS

Brute Force

Computer & Electrical Engineering and Computer Science Dept.
Florida Atlantic University

Spring 2017

Brute Force



Reading assignment :

- Anany Levitin, Introduction to The Design & Analysis of Algorithms, 2nd edition, Addison Wesley, 2007.
  - Chapter 3: Brute Force
  - Chapter 5.4: Algorithms for generating combinatorial objects

Brute Force

- Straight forward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved
- Proceeds in a simple and obvious way, but will require a huge number of steps to complete

Brute Force

- Applicable to a large variety of problems
- For some problems, brute-force approach yields reasonable algorithms
- Can be used if only few instances of the problem need to be solved
  - Avoids the expense of designing a more efficient algorithm
- Can be useful for solving small-size instances of a problem
- Can be used as a yardstick to compare more efficient alternatives for solving a problem

Brute-force algorithms

- Selection Sort
- Bubble Sort (see lecture 1)
- String Matching
- Closest-Pair
- Exhaustive Search
  - Traveling Salesman Problem
  - Knapsack Problem
  - Assignment Problem
  - Independent Set Problem

Selection Sort

- Scan the array 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 element.
- Generally, on the pass *i* ($0 \le i \le n$-2), find the smallest element in *A*[*i*..*n*-1] and swap it with *A*[i]

$$A_0 \le A_1 \le \ldots \le A_{i-1} \mid A_i, \ldots, A_{\min}, \ldots, A_{n-1}$$

in their final position   the last n-i elements

- After n-1 passes, the list is sorted

Selection Sort, example

<table>
<tr><td>| 89</td><td>45</td><td>68</td><td>90</td><td>29</td><td>34</td><td>17</td></tr>
<tr><td>17 |</td><td>45</td><td>68</td><td>90</td><td>29</td><td>34</td><td>89</td></tr>
<tr><td>17</td><td>29 |</td><td>68</td><td>90</td><td>45</td><td>34</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34 |</td><td>90</td><td>45</td><td>68</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45 |</td><td>90</td><td>68</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45</td><td>68 |</td><td>90</td><td>89</td></tr>
<tr><td>17</td><td>29</td><td>34</td><td>45</td><td>68</td><td>89 |</td><td>90</td></tr>
</table>

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 $A[min]$

RT analysis:

T(n) = (n-1) + (n-2) + … + 1 = (n-1)n/2 = $\Theta(n^2)$

Brute-Force String Matching

- *pattern*: a string of *m* characters to search for
- *text*: a (longer) string of *n* characters to search in
- problem: find a substring in the text that matches the pattern

$$t_0 \ldots t_i \ldots t_{i+j} \ldots t_{i+m-1} \ldots t_{n-1} \qquad \text{text T}$$

$$p_0 \ldots p_j \ldots p_{m-1} \qquad \text{pattern P}$$

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

Examples

1. **Pattern**: 001011
   Text: 10010101101001100101111010

2. **Pattern**: algorithm
   Text: The established framework for analyzing an algorithm's time efficiency is primarily grounded in the order of growth of the algorithm's running time as its input size goes to infinity.

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 $i \leftarrow 0$ **to** $n - m$ **do**
        $j \leftarrow 0$
        while $j < m$ **and** $P[j] = T[i + j]$ **do**
            $j \leftarrow j + 1$
        if $j = m$ **return** i
    return -1

- RT = O(nm)

Closest Pair

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

Brute-force algorithm

- Compute the distance between every pair of distinct points
- Return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm

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 $index1, index2$

- RT=O(n^2)

Brute-Force Strengths and Weaknesses

- Strengths
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g. sorting, searching, string matching)

- Weaknesses
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

Exhaustive Search

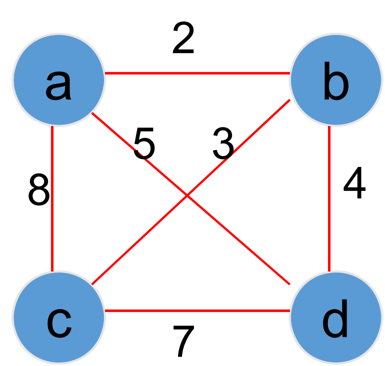
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

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

Example 1: Traveling Salesman 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
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



How do we represent a solution (Hamiltonian circuit)?

TSP by Exhaustive Search

Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

so on…

Efficiency:

- Assuming the start city is given, $(n-1)!$ tours
- RT = $\Theta(n(n-1)!) = \Theta(n!)$

Example 2: Knapsack Problem

Given n items:

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

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

Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Example 2: Knapsack Problem

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	**15**	**$80**
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Number of subsets is $2^n \Rightarrow T(n) = \Theta(n \cdot 2^n)$

Example 3: 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.

	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

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

How many assignments are there?

Assignment Problem by Exhaustive Search

How many assignments are there?

- Each feasible assignment is an *n*-tuple <j_1, j_2, …, j_n> where j_i is the job number assigned to the i^{th} person
- Example:

  <2, 3, 4, 1> – person 1 gets job 2, person 2 gets job 3, so on

- The number of assignments is $n!$
- $T(n) = \Theta(n \cdot n!)$

Assignment Problem by Exhaustive Search

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

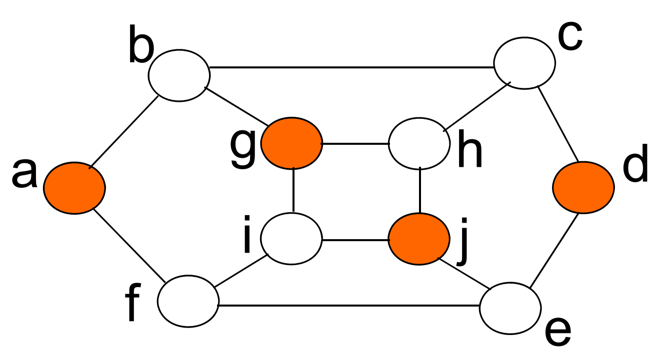
Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
etc.	

(For this particular instance, the optimal assignment is: 2, 1, 3, 4)

Example 4: k-Independent Set Problem

- K-Independent Set problem: *Given a graph G with n nodes, find whether G has an* ***independent set*** *of size k.*

A set S of nodes in G, $S \subseteq V$, is <u>independent</u> if no two nodes in S are joined by an edge.



S = {a, g, j, d} is an independent set of size 4

k-Independent Set Problem

- brute force algorithm:

```
for each subset S of k nodes
    check if S is an independent set
        if S is an independent set
            return TRUE
return FALSE
```

- The number of subsets of k nodes is $\binom{n}{k} = \theta(n^k)$
- To check if a subset of k vertices is independent takes

$$\binom{k}{2} = \theta(k^2)$$

- total RT $=\Theta(n^k k^2)$

- If k is constant, then RT = $\Theta(n^k)$

Example 5: Independent Set Problem

- Independent Set problem: *Given a graph G with n nodes, find an independent set of maximum size*

- brute force algorithm:

```
for each subset S of nodes
     check if S is an independent set
          if S is an independent set and |S| is larger than
                                   the max size so far
          then record |S| as the max-size set
return the max-size set
```

RT = $\Theta(2^n n^2)$

Remarks on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time <u>only on very small instances</u>
- Usually, there are much better alternatives!
- For some problems, exhaustive search or its variation is the only known way to get exact solution

Algorithms for Generating Combinatorial Objects

- Generating Permutations
- Generating Subsets

Generating Permutations

- Goal: generate n! permutations of {1, 2, …n}
- Decrease-by-one technique:
  - Assume that we have solved the smaller-by-one problem: generate all (n-1)! permutations
  - Insert n in each of the n possible positions among elements of every permutation of n-1 elements

  $\Rightarrow$ n! permutations obtained

Generating Permutations

- Bottom-up minimal change algorithm
  - **Minimal-change** requirement: each permutation can be obtained from its immediate predecessor by exchanging just two elements in it
  - *n* can be inserted in previously generated permutations either left-to-right or right-to-left
    - one way: insert *n* into 12…(*n*-1) by moving right-to-left and then switch direction each time a new permutation {1, 2, …, *n*-1} has to be processed

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

Generating permutations bottom up

Generating Permutations

- Johnson-Trotter algorithm
  - Same ordering of permutations of *n* elements without explicitly generating permutations for smaller *n*
  - Associate a direction with each element *k* in the permutation:

$$\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}$$

  - The element *k* is **mobile** if its arrow points to a smaller number adjacent to it
    - 3 and 4 are mobile, 2 and 1 are not

Generating Permutations

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 and the adjacent integer k's arrow points to
        reverse the direction of all the elements that are larger than k
        add the new permutation to the list

- RT = $\Theta(n!)$
- Example for n = 3 (largest mobile highlighted)

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

Generating Subsets

- Let A = $\{a_1, a_2, \ldots, a_n\}$
- There are 2^n subsets of A
- **Power set** = the set of all subsets
- Decrease-by-one technique:
  - Find a list of all subsets of $\{a_1, a_2, \ldots, a_{n-1}\}$
  - Then add to the list all the elements with a_n in each of them
  - Example for $\{a_1, a_2, a_3\}$

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

Generating subsets bottom up

Generating Subsets

- Bit string approach:
  - One-to-one correspondence between all 2^n subsets of an *n*-element set $\{a_1, a_2, \ldots, a_n\}$ and all 2^n bit strings $b_1b_2\ldots b_n$ of length *n*
  - Each binary string corresponds to a subset:
    - if $b_i = 1$, then $a_i \in$ subset; if $b_i = 0$, then $a_i \notin$ subset
  - Generate all the bit strings of length *n* by generating successive binary numbers from 0 to 2^n-1
    - Then map to the corresponding subsets
  - Example for n = 3:

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