Topic 2

Brute Force

Brute Force

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

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

1. Brute-Force Sorting Algorithm

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.
- Do this for all elements (until the second last).

$\mathbf{0 \leq \mathit{i} \leq \mathit{n}\text{-}2}$

$A[0] \leq \; . \;\; . \;\; . \; \leq A[i\text{-}1] \mid A[i], \; . \;\; . \;\; . \; , A[min], \; . \;\; . \;\; ., A[n\text{-}1]$

Example: 7 3 2 5

Analysis of 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 nondecreasing 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 $A[min]$

As an example, the action of the algorithm 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

Analysis of Selection Sort

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

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

$$= \frac{(n-1)n}{2}.$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Time efficiency: **$\Theta(n\text{^}2)$**

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

$$
\begin{array}{cccccccccc}
t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & \ldots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & \\
 & & p_0 & \ldots & p_j & \ldots & p_{m-1} & & \text{pattern } P &
\end{array}
$$

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 of Brute-Force String Matching

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

2. **Pattern:** happy
   Text: It is never too late to have a happy life.

Pseudocode and Efficiency

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
```

Time efficiency: **$\Theta(mn)$ comparisons (in the worst case)**

Why?

Ans: For each (n-m+1) tries, we compare m times at maximum.

→ $\Theta(mn)$ comparisons (in the worst case)

3. Brute-Force 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 $x = x_0$

<u>Brute-force algorithm</u>

$p \leftarrow 0.0$
for $i \leftarrow n$ **downto** 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

Efficiency: **$\Sigma_{0 \le i \le n} (i+1) = \Theta(n^2)$ multiplications**

Polynomial Evaluation: Improvement

We can do better by evaluating from right to left:

<u>Better brute-force algorithm</u>

$p \leftarrow a[0]$
$power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $power \leftarrow power * x$
  $p \leftarrow p + a[i] * power$
return p

Efficiency: **$\Theta(n)$ multiplications**

Horner's Rule is another linear time method.

4. Closest-Pair Problem

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

<u>Brute-force algorithm</u>

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.

Closest-Pair Brute-Force Algorithm (cont.)

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

Basic Operation: Square root (a complicated operation) and multiplication

Can you avoid the square root in the loop?

Efficiency:

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

$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2)$$

Divide-and-conquer approach can make it faster!

5. 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

6. Exhaustive Search

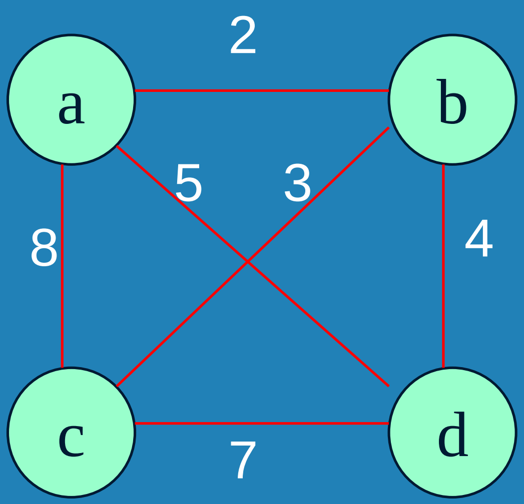
A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

Method:

- generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 5.4)
- 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

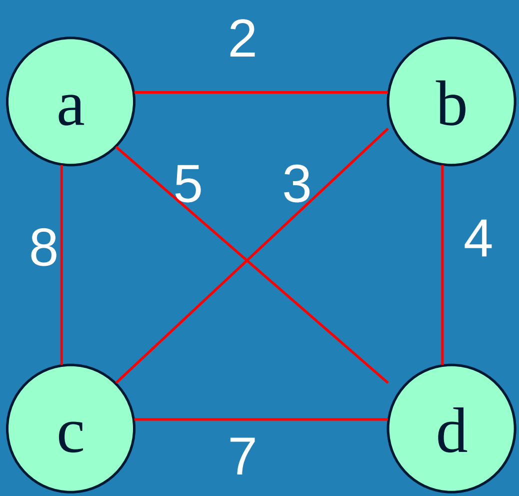
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



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

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



Efficiency: $\Theta((n-1)!)$

Chapter 5 discusses how to generate permutations fast.

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

Knapsack Problem by Exhaustive Search

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

Efficiency: $\Theta(2^n)$

Each subset can be represented by a binary string (bit vector, Ch 5).

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 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

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

How many assignments are there? n!

Assignment Problem by Exhaustive Search

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
2,1,3,4	6+2+1+4 = 13
2,1,4,3	…
2,3,1,4	
2,3,4,1	
2,4,1,3	
2,4,3,1	
…	

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4

- One can pick the best person for the job by looking at the minimum in each column.
- This can reduce the search space.

Number of possible assignment = 24 = 4!

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time <u>only on very small instances</u>

- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem

The Hungarian method runs in O(n^3) time.

- In many cases, exhaustive search or its variation is the only known way to get exact solution