Brute Force

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

- Examples:
  
  Computing $a^n$ ($a > 0$, $n$ a nonnegative integer) by repeated multiplication
  
  Computing $n!$ by repeated multiplication
  
  Multiplying two matrices following the definition
  
  Searching for a key in a list sequentially
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

- **Pattern:** 001011  
  **Text:** 10010101101001100101111010

- **Pattern:** happy  
  **Text:** It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

- Pattern: 001011
  Text: 10010101101001100101111010

- Pattern: happy
  Text: It is never too late to have a happy childhood.
Examples of Brute-Force String Matching

• Pattern: 001011
  Text: 10010101101001100101111010

• Pattern: happy
  Text: It is never too late to have a happy childhood.
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
Pseudocode and Efficiency

**ALGORITHM** \( \text{BruteForceStringMatch}(T[0..n-1], P[0..m-1]) \)

//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
    \( j \leftarrow 0 \)
    \( \text{while } j < m \) and \( P[j] = T[i+j] \) do
        \( j \leftarrow j + 1 \)
    if \( j = m \) return \( i \)
return \(-1\)

**Efficiency:**  A: \( O(n) \)  B: \( O(m(n-m)) \)  C: \( O(m) \)  D: \( O(m^2) \)
Brute-Force Polynomial Evaluation
Brute-Force Polynomial Evaluation

- Problem: Find the value of polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \] at a point \( x = x_0 \)
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 \( x = x_0 \)

• Brute-force algorithm

\[
\begin{align*}
p & \leftarrow 0.0 \\
\text{for } i & \leftarrow n \text{ downto } 0 \text{ do} \\
& \quad \text{power } \leftarrow 1 \\
& \quad \text{for } j \leftarrow 1 \text{ to } i \text{ do} \quad // \text{compute } x^i \\
& \quad \quad \text{power } \leftarrow \text{power } \ast x \\
& \quad \quad p \leftarrow p + a[i] \ast \text{power} \\
\text{return } p
\end{align*}
\]
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 \( x = x_0 \)

- Brute-force algorithm
  
  ```
  p ← 0.0
  for i ← n downto 0 do
    power ← 1
    for j ← 1 to i do //compute \( x^i \)
      power ← power * x
    p ← p + a[i] * power
  return p
  ```

- Efficiency:
Brute-Force Polynomial Evaluation

• Problem: Find the value of polynomial

\[ p(x) = a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0 \] at a point \( x = x_0 \)

• Brute-force algorithm

\[
\begin{align*}
\text{p} & \leftarrow 0.0 \\
\text{for } i \leftarrow n \text{ downto } 0 \text{ do} \\
& \quad \text{power} \leftarrow 1 \\
& \quad \text{for } j \leftarrow 1 \text{ to } i \text{ do} \quad \text{//compute } x^i \\
& \quad & \quad \text{power} \leftarrow \text{power} \ast x \\
& \quad & \quad \text{p} \leftarrow \text{p} + a[i] \ast \text{power} \\
\text{return } \text{p}
\end{align*}
\]

• Efficiency: A: \( O(n) \)  B: \( O(n^2) \)  C: \( O(lg n) \)  D: \( O(n^3) \)
Polynomial Evaluation: Improvement
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:

\[
p \leftarrow a[0] \\
\text{power} \leftarrow 1 \\
\text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad \text{power} \leftarrow \text{power} \times x \\
\quad p \leftarrow p + a[i] \times \text{power} \\
\text{return } p
\]
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:

\[
\begin{align*}
p & \leftarrow a[0] \\
power & \leftarrow 1 \\
\text{for } i & \leftarrow 1 \text{ to } n \text{ do} \\
& \quad \text{power } \leftarrow \text{power } \times x \\
& \quad p \leftarrow p + a[i] \times \text{power} \\
\text{return } p
\end{align*}
\]
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:

```plaintext
p ← a[0]
power ← 1
for i ← 1 to n do
    power ← power * x
    p ← p + a[i] * power
return p
```

• Efficiency:
Polynomial Evaluation: Improvement

• We can do better by evaluating from right to left:

• Better brute-force algorithm:

\[
\begin{align*}
p & \leftarrow a[0] \\
\text{power} & \leftarrow 1 \\
\text{for } i & \leftarrow 1 \text{ to } n \text{ do} \\
\quad & \text{power} \leftarrow \text{power} \ast x \\
\quad & p \leftarrow p + a[i] \ast \text{power} \\
\text{return } p
\end{align*}
\]

• Efficiency:  A: $O(n)$  B: $O(n^2)$  C: $O(lg\ n)$  D: $O(n^3)$
Closest-Pair Problem

• 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
    ◦ and return the indices of the points for which the distance is the smallest.
ALGORITHM \( \text{BruteForceClosestPoints}(P) \)

//Finds 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: Indices index1 and index2 of the closest pair of points
\( d_{\text{min}} \leftarrow \infty \)
\textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n - 1 \) \textbf{do}
\hspace{1em} \textbf{for} \( j \leftarrow i + 1 \) \textbf{to} \( n \) \textbf{do}
\hspace{2em} \( d \leftarrow \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \) //\( \sqrt{\text{sqrt}} \) is the square root function
\hspace{2em} \textbf{if} \( d < d_{\text{min}} \)
\hspace{3em} \( d_{\text{min}} \leftarrow d; \text{index1} \leftarrow i; \text{index2} \leftarrow j \)
\textbf{return} index1, index2
Closest-Pair Brute-Force Algorithm (cont.)

**ALGORITHM**  
`BruteForceClosestPoints(P)`

// Finds two closest points in the plane by brute force
// Input: A list `P` of `n` (`n ≥ 2`) points `P_i = (x_i, y_i), ..., P_n = (x_n, y_n)`
// Output: Indices `index1` and `index2` of the closest pair of points

```plaintext
d_{min} ← \infty
for i ← 1 to n - 1 do
    for j ← i + 1 to n do
        d ← sqrt((x_i - x_j)^2 + (y_i - y_j)^2) // `sqrt` is the square root function
        if d < d_{min}
            d_{min} ← d; index1 ← i; index2 ← j
return index1, index2
```

- **Efficiency:**
Closest-Pair Brute-Force Algorithm (cont.)

**ALGORITHM**  \( \text{BruteForceClosestPoints}(P) \)

//Finds 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: Indices \( \text{index1} \) and \( \text{index2} \) of the closest pair of points

\[
d_{\text{min}} \leftarrow \infty
\]

\[
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do}
\]

\[
\text{for } j \leftarrow i + 1 \text{ to } n \text{ do}
\]

\[
d \leftarrow \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad \text{//} \sqrt{\text{ is the square root function}}
\]

\[
\text{if } d < d_{\text{min}}
\]

\[
d_{\text{min}} \leftarrow d; \text{index1} \leftarrow i; \text{index2} \leftarrow j
\]

\[
\text{return index1, index2}
\]

- **Efficiency:**  \( \text{A: O(n)} \)  \( \text{B: O(n^2)} \)  \( \text{C: O(lg n)} \)  \( \text{D: O(n^3)} \)
Closest-Pair Brute-Force Algorithm (cont.)

**ALGORITHM** \texttt{BruteForceClosestPoints}(P)

//Finds 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: Indices \texttt{index1} and \texttt{index2} of the closest pair of points
\[ d_{\text{min}} \leftarrow \infty \]
\[
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do }
\]
\[
\text{for } j \leftarrow i + 1 \text{ to } n \text{ do }
\]
\[
d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2) \quad \text{//sqrt is the square root function}
\]
\[
\text{if } d < d_{\text{min}}
\]
\[
d_{\text{min}} \leftarrow d; \texttt{index1} \leftarrow i; \texttt{index2} \leftarrow j
\]
\[
\text{return } \texttt{index1}, \texttt{index2}
\]

- **Efficiency:** A: O(n)  B: O(n^2)  C: O(lg n)  D: O(n^3)

- **How to make it faster?**
Problem:

If \( \sqrt{\text{ }} \) is 10 x slower than \( \times \) and +, by how much will \( \text{BruteForceClosestPoints} \) speed up when we take out the \( \sqrt{\text{ }} \)?

A. \( \sim 10 \) times
B. \( \sim 100 \) times
C. \( \sim 1000 \) times
Problem:

Can you design a more efficient algorithm than the one based on the brute-force strategy to solve the closest-pair problem for \( n \) points \( x_1, \ldots, x_n \) on the real line?
Brute Force Closest Pair

• An Example of a particular kind of Brute Force Algorithm based on:

Exhaustive search
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. 4.3)
  ▶ 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:
TSP by Exhaustive Search

<table>
<thead>
<tr>
<th>Tour</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a→b→c→d→a</td>
<td>2+3+7+5 = 17</td>
</tr>
<tr>
<td>a→b→d→c→a</td>
<td>2+4+7+8 = 21</td>
</tr>
<tr>
<td>a→c→b→d→a</td>
<td>8+3+4+5 = 20</td>
</tr>
<tr>
<td>a→c→d→b→a</td>
<td>8+7+4+2 = 21</td>
</tr>
<tr>
<td>a→d→b→c→a</td>
<td>5+4+3+8 = 20</td>
</tr>
<tr>
<td>a→d→c→b→a</td>
<td>5+7+3+2 = 17</td>
</tr>
</tbody>
</table>

More tours?
Less tours?
Efficiency:

![Graph with nodes and edges showing the tours with their respective costs.](image)
TSP by Exhaustive Search

<table>
<thead>
<tr>
<th>Tour</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a→b→c→d→a</td>
<td>2+3+7+5 = 17</td>
</tr>
<tr>
<td>a→b→d→c→a</td>
<td>2+4+7+8 = 21</td>
</tr>
<tr>
<td>a→c→b→d→a</td>
<td>8+3+4+5 = 20</td>
</tr>
<tr>
<td>a→c→d→b→a</td>
<td>8+7+4+2 = 21</td>
</tr>
<tr>
<td>a→d→b→c→a</td>
<td>5+4+3+8 = 20</td>
</tr>
<tr>
<td>a→d→c→b→a</td>
<td>5+7+3+2 = 17</td>
</tr>
</tbody>
</table>

More tours?
Less tours?
Efficiency:

A: O(n)
B: O(n^2)
C: O(n^3)
D: O((n-1)!)  
E: O(n!)
Example 2: Knapsack Problem

- Given n items:
  - weights: \( w_1 \), \( w_2 \), \ldots, \( w_n \)
  - values: \( v_1 \), \( v_2 \), \ldots, \( v_n \)
  - a knapsack of capacity \( W \)

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

- Example: Knapsack capacity \( W=16 \)

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>2.</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>3.</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>4.</td>
<td>5</td>
<td>$10</td>
</tr>
</tbody>
</table>
Knapsack Problem by Exhaustive Search

<table>
<thead>
<tr>
<th>Subset</th>
<th>Total weight</th>
<th>Total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>{2}</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>{3}</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>$10</td>
</tr>
<tr>
<td>{1,2}</td>
<td>7</td>
<td>$50</td>
</tr>
<tr>
<td>{1,3}</td>
<td>12</td>
<td>$70</td>
</tr>
<tr>
<td>{1,4}</td>
<td>7</td>
<td>$30</td>
</tr>
<tr>
<td>{2,3}</td>
<td>15</td>
<td>$80</td>
</tr>
<tr>
<td>{2,4}</td>
<td>10</td>
<td>$40</td>
</tr>
<tr>
<td>{3,4}</td>
<td>15</td>
<td>$60</td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,4}</td>
<td>12</td>
<td>$60</td>
</tr>
<tr>
<td>{1,3,4}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{2,3,4}</td>
<td>20</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>22</td>
<td>infeasible</td>
</tr>
</tbody>
</table>

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

Knapsack capacity W=16
Knapsack Problem by Exhaustive Search

<table>
<thead>
<tr>
<th>Subset</th>
<th>Total weight</th>
<th>Total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>{2}</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>{3}</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>$10</td>
</tr>
<tr>
<td>{1,2}</td>
<td>7</td>
<td>$50</td>
</tr>
<tr>
<td>{1,3}</td>
<td>12</td>
<td>$70</td>
</tr>
<tr>
<td>{1,4}</td>
<td>7</td>
<td>$30</td>
</tr>
<tr>
<td>{2,3}</td>
<td>15</td>
<td>$80</td>
</tr>
<tr>
<td>{2,4}</td>
<td>10</td>
<td>$40</td>
</tr>
<tr>
<td>{3,4}</td>
<td>15</td>
<td>$60</td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,4}</td>
<td>12</td>
<td>$60</td>
</tr>
<tr>
<td>{1,3,4}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{2,3,4}</td>
<td>20</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>22</td>
<td>infeasible</td>
</tr>
</tbody>
</table>

Knapsack capacity $W = 16$

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>2.</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>3.</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>4.</td>
<td>5</td>
<td>$10</td>
</tr>
</tbody>
</table>

Portland State University
## Knapsack Problem by Exhaustive Search

<table>
<thead>
<tr>
<th>Subset</th>
<th>Total weight</th>
<th>Total value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>{2}</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>{3}</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>{4}</td>
<td>5</td>
<td>$10</td>
</tr>
<tr>
<td>{1,2}</td>
<td>7</td>
<td>$50</td>
</tr>
<tr>
<td>{1,3}</td>
<td>12</td>
<td>$70</td>
</tr>
<tr>
<td>{1,4}</td>
<td>7</td>
<td>$30</td>
</tr>
<tr>
<td>{2,3}</td>
<td>15</td>
<td>$80</td>
</tr>
<tr>
<td>{2,4}</td>
<td>10</td>
<td>$40</td>
</tr>
<tr>
<td>{3,4}</td>
<td>15</td>
<td>$60</td>
</tr>
<tr>
<td>{1,2,3}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,4}</td>
<td>12</td>
<td>$60</td>
</tr>
<tr>
<td>{1,3,4}</td>
<td>17</td>
<td>infeasible</td>
</tr>
<tr>
<td>{2,3,4}</td>
<td>20</td>
<td>infeasible</td>
</tr>
<tr>
<td>{1,2,3,4}</td>
<td>22</td>
<td>infeasible</td>
</tr>
</tbody>
</table>

Knapsack capacity \( W = 16 \)

### Item List

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>$10</td>
</tr>
</tbody>
</table>

#### Efficiency?

- A: \( O(n^2) \)
- B: \( O(2^n) \)
- C: \( O(n!) \)
- D: \( O((n-1)!) \)
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 \( p \) to job \( j \) is \( C[i, j] \). Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

• How many assignments are there …
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 $p$ to job $j$ is $C[i, j]$. Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there ...
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 $p$ to job $j$ is $C[i, j]$. Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there …
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 $p$ to job $j$ is $C[i, j]$. Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th>Person 1</th>
<th>9</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there ...
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 \( p \) to job \( j \) is \( C[i, j] \). Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there …

an assignment \( \langle a_1, a_2, a_3, a_4 \rangle \) means that person \( i \) gets job \( a_i \)
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 $p$ to job $j$ is $C[i,j]$. Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there ...

- *an assignment* $\langle a_1, a_2, a_3, a_4 \rangle$ means that person $i$ gets job $a_i$
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 \( p \) to job \( j \) is \( C[i, j] \). Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there …
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 $p$ to job $j$ is $C[i, j]$. Find an assignment that minimizes the total cost.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person 1</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Person 2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Person 3</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Person 4</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

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

- How many assignments are there …
Assignment Problem by Exhaustive Search
Assignment Problem by Exhaustive Search

• Consider the problem in terms of the Cost Matrix $C$

$$C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4 \\
\end{bmatrix}$$
Assignment Problem by Exhaustive Search

Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
</tbody>
</table>

$C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4
\end{bmatrix}$
Assignment Problem by Exhaustive Search

• Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
</tbody>
</table>

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$
Assignment Problem by Exhaustive Search

- Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>9+3+8+6=26</td>
</tr>
</tbody>
</table>

$$C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4
\end{bmatrix}$$
Assignment Problem by Exhaustive Search

- Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>9+3+8+6=26</td>
</tr>
<tr>
<td>1, 4, 2, 3</td>
<td>9+7+8+9=33</td>
</tr>
</tbody>
</table>

$$
C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4
\end{bmatrix}
$$
Assignment Problem by Exhaustive Search

- Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>9+3+8+6=26</td>
</tr>
<tr>
<td>1, 4, 2, 3</td>
<td>9+7+8+9=33</td>
</tr>
<tr>
<td>1, 4, 3, 2</td>
<td>9+7+1+6=23</td>
</tr>
</tbody>
</table>

\[
C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4 \\
\end{bmatrix}
\]
Assignment Problem by Exhaustive Search

Consider the problem in terms of the **Cost Matrix** $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>$9+4+1+4=18$</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>$9+4+8+9=30$</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>$9+3+8+4=24$</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>$9+3+8+6=26$</td>
</tr>
<tr>
<td>1, 4, 2, 3</td>
<td>$9+7+8+9=33$</td>
</tr>
<tr>
<td>1, 4, 3, 2</td>
<td>$9+7+1+6=23$</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$
Assignment Problem by Exhaustive Search

Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>9+3+8+6=26</td>
</tr>
<tr>
<td>1, 4, 2, 3</td>
<td>9+7+8+9=33</td>
</tr>
<tr>
<td>1, 4, 3, 2</td>
<td>9+7+1+6=23</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

How many assignments are there?

$$C = \begin{bmatrix}
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4 \\
\end{bmatrix}$$
Assignment Problem by Exhaustive Search

- Consider the problem in terms of the Cost Matrix $C$

<table>
<thead>
<tr>
<th>Assignment (col.#s)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9+4+1+4=18</td>
</tr>
<tr>
<td>1, 2, 4, 3</td>
<td>9+4+8+9=30</td>
</tr>
<tr>
<td>1, 3, 2, 4</td>
<td>9+3+8+4=24</td>
</tr>
<tr>
<td>1, 3, 4, 2</td>
<td>9+3+8+6=26</td>
</tr>
<tr>
<td>1, 4, 2, 3</td>
<td>9+7+8+9=33</td>
</tr>
<tr>
<td>1, 4, 3, 2</td>
<td>9+7+1+6=23</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
</tr>
</tbody>
</table>

How many assignments are there?

A: $O(n)$  B: $O(n^2)$  C: $O(n^3)$  D: $O(n!)$
Convex Hulls

• What is a Convex Hull?

A. A bad design for a boat
B. A good design for a boat
C. A set of points without any concavities
D. None of the above
Convex Hulls

• What is a Convex Set?

A. A bad design for a boat
B. A good design for a boat
C. A set of points without any concavities
D. None of the above
Convex Hulls

• What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

• What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

• What is a Convex Set?
Convex Hulls

• What is a Convex Set?
Convex Hulls

• What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

• What is a Convex Set?
Convex Hulls

- What is a Convex Set?
Convex Hulls

A set of points $C$ is *convex* iff $\forall \ a, b \in C$, all points on the line segment $ab$ are entirely in $C$
Convex Hulls
Convex Hulls

• Given an arbitrary set of points $S$, the convex hull of $S$ is the smallest convex set that contain all the points in $S$. 
Convex Hulls

• Given an arbitrary set of points S, the convex hull of S is the smallest convex set that contain all the points in S.

  ▶ Barricading sleeping tigers
Convex Hulls

- Given an arbitrary set of points $S$, the convex hull of $S$ is the smallest convex set that contain all the points in $S$.
  - Barricading sleeping tigers
  - Rubber-band around nails
Applications of Convex Hull

- Collision-detection in video games
Applications of Convex Hull

- Collision-detection in video games
- Robot motion planning
Theorems about Convex Hulls

- The convex hull of a set $S$ is a convex polygon all of whose vertices are at some of the points of $S$.
- A line segment $ab$ is part of the boundary of the convex hull of $S$ iff all the points of $S$ lie on the same side of $ab$ (or on $ab$).
Theorems about Convex Hulls

• The convex hull of a set $S$ is a convex polygon all of whose vertices are at some of the points of $S$.

• A line segment $ab$ is part of the boundary of the convex hull of $S$ iff all the points of $S$ lie on the same side of $ab$ (or on $ab$)
Theorems about Convex Hulls

- The convex hull of a set $S$ is a convex polygon all of whose vertices are at some of the points of $S$.
- A line segment $ab$ is part of the boundary of the convex hull of $S$ iff all the points of $S$ lie on the same side of $ab$ (or on $ab$)
Theorems about Convex Hulls

- The convex hull of a set $S$ is a convex polygon all of whose vertices are at some of the points of $S$.

- A line segment $ab$ is part of the boundary of the convex hull of $S$ iff all the points of $S$ lie on the same side of $ab$ (or on $ab$).
Brute-Force Algorithm for Convex Hull

- write it down!
  - Assume that you have a method for ascertaining if a point \(r\) is on a line \(pq\), on the -ve side of line \(pq\), or on the +ve side of \(pq\)
Brute-Force Algorithm for Convex Hull

• write it down!
  ▶ Assume that you have a method for ascertaining if a point \( r \) is on a line \( pq \), on the –ve side of line \( pq \), or on the +ve side of \( pq \)

\[
r.\text{whichSideOfLine}(pq)
\]
Brute-Force Algorithm for Convex Hull

• write it down!
  ▶ Assume that you have a method for ascertaining if a point \( r \) is on a line \( pq \), on the –ve side of line \( pq \), or on the +ve side of \( pq \)

\[ r\.\text{whichSideOfLine}(pq) \]
Brute-Force Algorithm for Convex Hull

- write it down!
  - Assume that you have a method for ascertaining if a point \( r \) is on a line \( pq \), on the \(-ve\) side of line \( pq \), or on the \(+ve\) side of line \( pq \)

\[
\begin{align*}
\text{r.whichSideOfLine}(pq) \\
= \begin{cases} 
1 & \text{if } r \text{ is on the } +ve \text{ side of } pq \\
0 & \text{if } r \text{ is on the line } pq \\
-1 & \text{if } r \text{ is on the } -ve \text{ side of } pq
\end{cases}
\end{align*}
\]

\[
ax + by = c
\]

\[
c = pxqy - qxpy
\]
Brute-Force Algorithm for Convex Hull

\[
\text{edgeSet} \leftarrow \{\}
\]

P: for \( p \) in \( S \) do:

Q: for \( q \) in \( S \), \( q \neq p \) do:

\[
goodSide \leftarrow 0
\]

R: for \( r \) in \( S \), \( r \neq p \land r \neq q \) do:

\[
side \leftarrow r \cdot \text{whichSideOfLine}(pq)
\]

if \( side \neq 0 \) then

\[
\text{if} \ goodSide = 0 \text{ then } goodSide \leftarrow side
\]

\[
\text{if} \ goodSide \neq side \text{ then exit Q.}
\]

\[
\text{edgeSet} \leftarrow \text{edgeSet} \cup \{pq\}
\]
Final Comments on Exhaustive Search

- 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
- However, in many cases, exhaustive search (or a variation) is the only known way to find an exact solution.
Searching in Graphs

Exhaustively search a graph, by traversing the edges, visiting every node once

Two approaches:

- Depth-first search and
- Breadth-first search
ALGORITHM \textit{DFS}(G)
\begin{verbatim}
//Implements a depth-first search traversal of a given graph
//Input: Graph \(G = \langle V, E \rangle\)
//Output: Graph \(G\) with its vertices marked with consecutive integers
//    in the order they are first encountered by the DFS traversal
mark each vertex in \(V\) with 0 as a mark of being “unvisited”
\(\text{count} \leftarrow 0\)
\textbf{for} each vertex \(v\) in \(V\) \textbf{do}
    \textbf{if} \(v\) is marked with 0
        \(\text{dfs}(v)\)
\end{verbatim}
\hspace{1cm} \(\text{dfs}(v)\)
\begin{verbatim}
//visits recursively all the unvisited vertices connected to vertex \(v\)
//by a path and numbers them in the order they are encountered
//via global variable \textit{count}
\(\text{count} \leftarrow \text{count} + 1; \quad \text{mark} \ v \ \text{with} \ \text{count}\)
\textbf{for} each vertex \(w\) in \(V\) adjacent to \(v\) \textbf{do}
    \textbf{if} \(w\) is marked with 0
        \(\text{dfs}(w)\)
\end{verbatim}
Example

dfs(1)
Example

dfs(1)
dfs(2)
Example

dfs(1)
dfs(2)
dfs(7)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)
dfs(3)
Example

$dfs(1)$
$dfs(2)$
$dfs(7)$
$dfs(5)$
$dfs(4)$
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(4)
Example

\[dfs(1)\]
\[dfs(2)\]
\[dfs(7)\]
\[dfs(5)\]
\[dfs(4)\]
Example

\[ \text{dfs}(1) \]
\[ \text{dfs}(2) \]
\[ \text{dfs}(7) \]
\[ \text{dfs}(5) \]
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(6)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
dfs(6)
Example

dfs(1)
dfs(2)
dfs(7)
dfs(5)
Example

dfs(1)
dfs(2)
dfs(7)
Example

dfs(1)
dfs(2)
Example

$dfs(1)$
Example

dfs(8)
Example

dfs(8)
dfs(9)
Example

dfs(8)
Example
Complexity?

• What's the basic operation?
  ▶ finding all the Vertices in the graph?
  ▶ making a mark?
  ▶ checking a mark?
  ▶ finding all the neighbors of a node?

• Cost depends on the data structure used to represent the graph
Two choices of data structure:

- Adjacency Matrix: $\Theta(|V|^2)$
- Adjacency List: $\Theta(|V| + |E|)$
One Last look at the Example
One Last look at the Example
One Last look at the Example

Blue edges form a spanning three
One Last look at the Example

Blue edges form a spanning three

Black edges lead back to an already-visited node
Applications

• Checking for connectivity
  ▶ How?

• Checking for Cycles
  ▶ How?