03 - Brute Force Algorithm

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 21-25 February 2022
Table of contents

- Principal of brute force algorithm
- Some examples of brute-force technique
  1. Finding max/min of array
  2. Sequential search
  3. Computing power
  4. Computing factorial
  5. Square-matrix multiplication
  6. Prime-number test
  7. Polynomial interpolation
  8. Closest pair problem
  9. Pattern matching

- Characteristics of brute force algorithm
- Exhaustive search (principal & examples)
  1. The Traveling Salesman Problem
  2. The 1/0 Knapsack Problem
  3. Exhaustive search in cryptography
- Exercises: Assignment problem; Partition problem; Magic square
- Heuristic technique (principal & examples)
Brute Force Algorithm
Definition (Brute Force algorithm or Exhaustive Search)

It is a typical problem-solving technique that uses straightforward approach.

The solution is uncovered by checking every possible answer one by one, by determining whether the result satisfies the statement of a problem or not.

The algorithm is usually based on:

- problem statement;
- definitions/concepts that are involved in the problem

Characteristics: simple, direct approach, obvious way
Given an integer $n$, to find all divisors of $n$, one could check whether each integer $i \in [1, n]$ can divide $n$.

Given a lock of 4-digit PIN, where the digits to be chosen from 0-9. The brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN (there are at most 10000 trials).
1. Finding the max/min element of an array

Problem. Given an array of $n$ integers $(a_1, a_2, \ldots, a_n)$. We want to find the maximum of the array.
1. Finding the max/min element of an array

**Problem.** Given an array of $n$ integers ($a_1, a_2, \ldots, a_n$). We want to find the maximum of the array.

**Brute-force approach:** to find the max, compare each element from $a_1$ to $a_n$.

---

**Algorithm 2** Finding maximum of an array of integers

1: procedure $\text{Max}(A[1..n])$
2: \hspace{1em} max $\leftarrow a_1$
3: \hspace{1em} for $i = 2$ to $n$ do
4: \hspace{2em} if $a_i > \text{max}$ then
5: \hspace{3em} \hspace{1em} max $\leftarrow a_i$
6: \hspace{2em} end if
7: \hspace{1em} end for
8: end procedure
1. Finding the max/min element of an array

**Problem.** Given an array of $n$ integers ($a_1, a_2, \ldots, a_n$). We want to find the maximum of the array.

**Brute-force approach:** to find the max, compare each element from $a_1$ to $a_n$.

**Algorithm 3** Finding maximum of an array of integers

```plaintext
1: procedure Max(A[1..n])
2: max ← $a_1$
3: for $i = 2$ to $n$ do
4: if $a_i >$ max then
5: max ← $a_i$
6: end if
7: end for
8: end procedure
```

**Complexity?** $O(n)$
Problem. Given an array of integers \((a_1, a_2, \ldots, a_n)\). We want to find an element \(x\) in the array. If \(x\) is found, the algorithm outputs the index of the element in the array. Otherwise, it outputs \(-1\).

**Brute-force approach:** compare each element in the array with \(x\). We are done if \(x\) is found or all elements are checked.
### Algorithm 4 Finding an element in an array of integers

1: **procedure** `SeqSearch`(A[1..n], x)
2: \( i \leftarrow 1 \)
3: **while** \( i < n \) and \( a_i \neq x \) **do**
4: \( i \leftarrow i + 1 \)
5: **end while**
6: **if** \( a_i = x \) **then**
7: \( \text{idx} \leftarrow i \)
8: **else**
9: \( \text{idx} \leftarrow -1 \)
10: **end if**
11: **return** \( \text{idx} \)
12: **end procedure**

**Complexity?** in the assignment!
Problem. Compute $a^n$ ($a > 0$, $n$ is a non-negative numbers).
Problem. Compute $a^n$ ($a > 0$, $n$ is a non-negative numbers).

Brute-force approach:

$$a^n = a \times a \times \cdots \times a \quad n \text{ times}$$

We multiply 1 with $a$ $n$ times.

Algorithm 6 Computing $a^n$

1: procedure POWER($a$, $n$)
2: \hspace{1em} result $\leftarrow$ 1
3: \hspace{1em} for $i = 1$ to $n$ do
4: \hspace{2em} result $\leftarrow$ result * $a$
5: \hspace{1em} end for
6: \hspace{1em} return result
7: end procedure
3. Powering (2)

Algorithm 5 Computing $a^n$

1: procedure $\text{POWER}(a, n)$
2: \hspace{1em} result $\leftarrow 1$
3: \hspace{1em} for $i = 1$ to $n$ do
4: \hspace{2em} result $\leftarrow$ result $\times a$
5: \hspace{1em} end for
6: \hspace{1em} return result
7: end procedure

Time complexity: $\mathcal{O}(n)$.
Can you explain why?

Is there a better algorithm for "power"?
Problem. Compute $n!$ ($n > 0$, $n$ is a non-negative integers).

Brute-force approach:

$$n! = 1 \times 2 \times \cdots \times n \text{ and } 0! = 1$$

We multiply the integers 1, 2, until $n$
4. Computing factorial (1)

Problem. Compute $n!$ ($n > 0$, $n$ is a non-negative integers).

Brute-force approach:

$$n! = 1 \times 2 \times \cdots \times n$$

and

$$0! = 1$$

We multiply the integers 1, 2, until $n$

---

**Algorithm 8** Computing $n!$

1: procedure **FACTORIAL**($n$)
2: result ← 1
3: if $n \leq 1$ then return result
4: else
5: for $i = 2$ to $n$ do
6: result ← result * $i$
7: end for
8: end if
9: return result
10: end procedure
Algorithm 5 Computing $n!$

1: procedure FACTORIAL($n$)
2:   result ← 1
3:   if $n \leq 1$ then return result
4:   else
5:     for $i = 2$ to $n$ do
6:       result ← result * $i$
7:     end for
8:   end if
9:   return result
10: end procedure

Time complexity: $\mathcal{O}(n)$ (multiplying $n$ numbers)
Problem. Given two square matrices, of size $n \times n$. Find a way to multiply the two matrices!

![Figure: A square matrix](image)

Brute Force Algorithm
Let $A = [a_{ij}]$, $B = [b_{ij}]$ be $n \times n$ matrices, and $C = A \times B$.

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$
Let $A = [a_{ij}]$, $B = [b_{ij}]$ be $n \times n$ matrices, and $C = A \times B$.

\[ c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj} \]

\[
\begin{array}{cccc}
\hline
\hline
\hline
& \cdot & \cdot & \cdot \\
\hline
\hline
\hline
\end{array}
\times
\begin{array}{cccc}
\hline
\hline
\hline
\hline
\hline
\end{array}
=
\begin{array}{cccc}
\hline
\hline
\hline
\hline
\hline
\end{array}
\]

**Brute-force approach:** compute each element of $C$ one-by-one by multiplying the corresponding row of $A$ and column of $B$. 
Algorithm 9 Square matrix multiplication

1: procedure MatrixMult(A, B)
2:     for i ← 1 to n do
3:         for j ← 1 to n do
4:             C[i, j] ← 0
5:             for k ← 1 to n do
7:             end for
8:         end for
9:     end for
10:    return C
11: end procedure
Time complexity: $O(n^3)$.

We look at the "dominant operations" as follows:

- Inside the inner-most loop: $n^3$ multiplications, $n^3$ additions, and $3$ assignments
- Inside the second loop: $n^2$ assignments
Problem. Given a positive integer $n$. Check if $n$ is prime.

Remark. A number $n$ is prime iff the only divisors of $n$ are 1 and $n$. 
Problem. Given a positive integer $n$. Check if $n$ is prime.

Remark. A number $n$ is *prime* iff the only divisors of $n$ are 1 and $n$.

1. **Brute-force approach:** divide $n$ by 2, 3, …, $n - 1$. If none of them divides $n$, then $n$ is a prime number.
6. Prime number test (2)

**Algorithm 10 Prime number test**

1: procedure IsPrime(n)
2:   if n < 2 then
3:     return False
4:   else
5:     isprime ← True;  k ← 2
6:     while isprime & (k ≤ n − 1) do
7:       if n mod k == 0 then
8:         isprime ← False
9:       else
10:          k ← k + 1
11:     end if
12:   end while
13:   return isprime
14: end if
15: end procedure
Problem. Given a positive integer $n$. Check if $n$ is prime.

Remark. A number $n$ is prime iff the only divisors of $n$ are 1 and $n$.

1. **Brute-force approach**: divide $n$ by 2, 3, \ldots, $n-1$. If none of them divides $n$, then $n$ is a prime number.

2. **Sieve of Eratosthenes**: for a given upper limit $n$, iteratively mark the multiples of primes as composite (i.e. not prime), starting from 2. Once all multiples of 2 have been marked composite, the multiples of next prime, i.e. 3 are marked composite. This process continues until $p \leq \sqrt{n}$, where $p$ is a prime number. 

By this technique, to check that $n$ is prime, we only need to check if there is any prime number $\leq \sqrt{n}$ that divides $n$. 
Algorithm 11 Prime number test (sieve of Eratosthenes)

1:  procedure IsPrime2(n)
2:     if $n < 2$ then
3:         return False
4:     else
5:         isprime ← True;  $k ← 2$
6:         while isprime & ($k \leq \sqrt{n}$) do
7:             if $n \mod k == 0$ then
8:                 isprime ← False
9:             else
10:                 $k ← k + 1$
11:             end if
12:         end while
13:     return isprime
14: end procedure
6. Prime number test (5)

Time complexity:

Algorithm 8 Prime number test
1: procedure IsPrime(n)
2: if n < 2 then
3: return False
4: else
5: isprime ← True;  k ← 2
6: while test & (k ≤ n − 1) do
7: if n mod k == 0 then
8: isprime ← False
9: else
10: k ← k + 1
11: end if
12: end while
13: return test
14: end if
15: end procedure

Algorithm 9 Prime number test (sieve of Eratosthenes)
1: procedure IsPrime2(n)
2: if n < 2 then
3: return False
4: else
5: isprime ← True;  k ← 2
6: while test & (k ≤ √n) do
7: if n mod k == 0 then
8: isprime ← False
9: else
10: k ← k + 1
11: end if
12: end while
13: return test
14: end if
15: end procedure
6. Prime number test (5)

Time complexity:

- **Brute-force:** $\mathcal{O}(n)$
- **Sieve-Eratosthenes brute-force:** $\mathcal{O}(\sqrt{n})$
Problem. Evaluate the following polynomial at $x = t$:

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]
Problem. Evaluate the following polynomial at $x = t$:

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

**Brute-force approach:** each of $x^k$ is computed as in the "powering algorithm"; then multiply $x^k$ with $a_k$, and take the sum with the other terms.
Algorithm 12 Polynomial interpolation

1: procedure POLYOM(n, A[0..n], t)
2: \hspace{1em} p \leftarrow 0
3: \hspace{1em} for i \leftarrow n \text{ downto } 0 \text{ do}
4: \hspace{2em} power \leftarrow 1
5: \hspace{2em} for j \leftarrow 1 \text{ to } i \text{ do}
6: \hspace{3em} power \leftarrow power \ast t
7: \hspace{2em} end for
8: \hspace{1em} p \leftarrow p + a[i] \ast power
9: \hspace{1em} end for
10: \hspace{1em} return p
11: end procedure
Algorithm 13 Polynomial interpolation

1: procedure POLYNOM(n, A[0..n], t)
2: \hspace{1em} p \leftarrow 0
3: \hspace{1em} for i \leftarrow n \text{ downto } 0 \text{ do}
4: \hspace{2em} power \leftarrow 1
5: \hspace{2em} for j \leftarrow 1 \text{ to } i \text{ do}
6: \hspace{3em} power \leftarrow power \ast t
7: \hspace{2em} end for
8: \hspace{1em} p \leftarrow p + a[i] \ast power
9: \hspace{1em} end for
10: \hspace{1em} return p
11: end procedure

There are $\mathcal{O}\left(\frac{n(n-1)}{2}\right) + \mathcal{O}(n + 1)$ operations.

Time complexity: $\mathcal{O}(n^2)$. 
Problem. Given $n$ points in the 2 dimensional Euclidean space (i.e. Cartesian coordinate system). Find two points with the shortest distance.

The distance between two points $p_1(x_1, y_1)$ and $p_2 = (x_2, y_2)$ is given by:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
8. Closest pair problem (1)

**Problem.** Given $n$ points in the 2 dimensional Euclidean space (i.e. Cartesian coordinate system). Find two points with the shortest distance.

The distance between two points $p_1(x_1, y_1)$ and $p_2 = (x_2, y_2)$ is given by:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

**Brute-force approach:**

1. Compute the distance between every pair of points
2. Take the pair with the minimum distance
Algorithm 14 Finding the closest points

1: procedure CLOSEST_POINTS(p₁, p₂, ..., pₙ)
2:    dmin ← 999999
3:    for i ← 1 to n − 1 do
4:        for j ← i + 1 to n do
5:            d ← \[\sqrt{(pᵢ.x - pⱼ.x)^2 + (pᵢ.y - pⱼ.y)^2}\]
6:                if d < dmin then
7:                    dmin ← d
8:                    A ← pᵢ
9:                    B ← pⱼ
10:               end if
11:        end for
12:    end for
13:    return A and B
14: end procedure
8. Closest pair problem (3)

Algorithm 13 Finding the closest points

1: procedure CLOSESTPOINTS($p_1, p_2, \ldots, p_n$)
2:     dmin $\leftarrow$ 999999
3:     for $i \leftarrow 1$ to $n - 1$ do
4:         for $j \leftarrow i + 1$ to $n$ do
5:             $d \leftarrow \sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2}$
6:             if $d < dmin$ then
7:                 dmin $\leftarrow d$
8:                 $A \leftarrow p_i$
9:                 $B \leftarrow p_j$
10:         end if
11:     end for
12: end for
13: return $A$ and $B$
14: end procedure

Time complexity: $O(n^2)$
Problem. Given a string of length $n$ and a pattern of length $m$ with $m < n$. Find the location of the first character of the pattern in the string that matches with the pattern.

Example.

- **String:** NOBODY NOTICED HIM
- **Pattern:** NOT
Problem. Given a string of length $n$ and a pattern of length $m$ with $m < n$. Find the location of the first character of the pattern in the string that matches with the pattern.

Example.

- **String**: NOBODY NOTICED HIM
- **Pattern**: NOT

**Brute-force approach:**

1. Start with the first character of the string.
2. Start from the first character of the pattern, check if the pattern matches with some substring:
   - all characters match
   - there is a character that doesn’t match
3. If the pattern doesn’t match, we move to the right, and repeat Step 2.
9. Pattern matching (2)

Example 1.

```
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
```

Example 2.

- 10010101001110101010001
- 001011

10010101001110101010001 (Try it!)
9. Pattern matching (3)

1: procedure PatternMatching($P, T$)
2: \hspace{1em} $i \leftarrow 0$; \hspace{0.5em} found \leftarrow False
3: \hspace{1em} while ($i \leq n - m$) \& (not found) do
4: \hspace{2em} $j \leftarrow 1$
5: \hspace{2em} while ($j \leq m$) \& $P_j = T_{i+j}$ do
6: \hspace{3em} $j \leftarrow j + 1$
7: \hspace{2em} end while
8: \hspace{1em} if $j = m$ then \hspace{0.5em} found \leftarrow True
9: \hspace{1em} else \hspace{0.5em} $i \leftarrow i + 1$
10: \hspace{1em} end if
11: \hspace{1em} end while
12: \hspace{1em} if found then \hspace{0.5em} return $i + 1$
13: \hspace{1em} else \hspace{0.5em} return $-1$
14: \hspace{1em} end if
15: end procedure
9. Pattern matching (4)

**Time complexity**

1. **Worst case**
   - In each matching trial, we match all characters of the pattern with the character in the corresponding character of the string → $m$ steps
   - This is done for all possible positions in the string → $n - m + 1$ possibilities
   - The number of steps: $m(n = m + 1) \in O(nm)$

2. **Best case**
   - This happens when the pattern is found in the first $m$ positions of the string
   - In this case, we check all characters of the pattern
   - Complexity: $O(m)$
Strength of brute-force:

- This is not a powerful algorithm, but almost all problems can be solved using brute force algorithm.
- Simple and easy to understand
- Can be applied for many problems: searching, sorting, string matching, matrix multiplication, etc.
- It produces standard algorithms for computational tasks such as multiplication/addition of \( n \) numbers, finding max/min in an array.
Drawbacks of brute-force:

- The algorithm is not "smart", because it needs many computation and takes long time to proceed. For many real-world problems, the number of natural candidates is prohibitively large.
- Brute force algorithm is suitable for small instance, because it is simple and can be easily implemented.

Remark. This algorithm is often called as naive algorithm, and is used to compare with the other powerful algorithm.
Exhaustive search is simply a brute-force approach to combinatorial problems (permutation, combination, subsets, etc.).

Remark. Example of combinatorial problems are the Traveling Salesman Problem, Knapsack problem, etc.; and non-combinatorial problems are Powering problem, Square Matrix Multipication, etc.

Remark. In many references, exhaustive search is considered same as brute force.

Read the book of Anany Levitin, look at Section 3.4 (page 143)!
**Problem.** Given $n$ cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

**Remark.** We may always assume that the given input graph is a *complete graph* (i.e. every pair of vertices is joined by an edge). If as in the figure above the graph is not complete (for example, there is no edge between vertex 1 and 7), then we may assume that the edge $(1, 7)$ exists, but its weight is $\infty$. 
Hamiltonian cycle is a cycle that visits each vertex of the graph exactly once. The TSP problem is equivalent to finding the Hamiltonian cycle of the minimum weight.

Exhaustive search algorithm for TSP

1. Enumerate all Hamiltonian cycles of an \( n \)-vertex complete graph.
2. Evaluate the weight of every Hamiltonian cycle that is found in step 1.
3. Choose the Hamiltonian cycle with the minimum weight.

Exercise. Apply the algorithm above to the following graph!
1. Traveling Salesman Problem (3)

<table>
<thead>
<tr>
<th>No.</th>
<th>Traveling route</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$</td>
<td>$10 + 8 + 12 + 9 = 39$</td>
</tr>
<tr>
<td>2.</td>
<td>$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$</td>
<td>$10 + 8 + 12 + 4 = 34$</td>
</tr>
<tr>
<td>3.</td>
<td>$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$</td>
<td>$4 + 8 + 8 + 9 = 29$</td>
</tr>
<tr>
<td>4.</td>
<td>$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$</td>
<td>$4 + 12 + 8 + 10 = 34$</td>
</tr>
<tr>
<td>5.</td>
<td>$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$</td>
<td>$9 + 8 + 8 + 4 = 29$</td>
</tr>
<tr>
<td>6.</td>
<td>$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$</td>
<td>$9 + 12 + 8 + 10 = 39$</td>
</tr>
</tbody>
</table>
1. Traveling Salesman Problem (3)

<table>
<thead>
<tr>
<th>No.</th>
<th>Traveling route</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$</td>
<td>$10 + 8 + 12 + 9 = 39$</td>
</tr>
<tr>
<td>2.</td>
<td>$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$</td>
<td>$10 + 8 + 12 + 4 = 34$</td>
</tr>
<tr>
<td>3.</td>
<td>$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$</td>
<td>$4 + 8 + 8 + 9 = 29$</td>
</tr>
<tr>
<td>4.</td>
<td>$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$</td>
<td>$4 + 12 + 8 + 10 = 34$</td>
</tr>
<tr>
<td>5.</td>
<td>$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$</td>
<td>$9 + 8 + 8 + 4 = 29$</td>
</tr>
<tr>
<td>6.</td>
<td>$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$</td>
<td>$9 + 12 + 8 + 10 = 39$</td>
</tr>
</tbody>
</table>

The shortest route is given by:

- $a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$, of weight 29
- $a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$, of weight 29
Exhaustive search algorithm for TSP

1. Enumerate all Hamiltonian cycle of an \( n \)-vertex complete graph.
2. Evaluate the weight of every Hamiltonian cycle that is found in step 1.
3. Choose the Hamiltonian cycle with the minimum weight.

Discuss how to compute the complexity?
Time complexity of exhaustive search of TSP

- Up to shifting, the number of different Hamiltonian cycles on \( n \) vertices is: \( \frac{(n-1)!}{2} \) (use the "cyclic permutation formula" (https://www.wikiwand.com/en/Cyclic_permutation), and note that the solution set can be grouped into pairs where one is the reflection of the other).

- So, to solve TSP with exhaustive search, then we have to enumerate \( \frac{(n-1)!}{2} \) Hamiltonian cycles, compute their weights, and choose the cycle that has the minimum weight.

- To compute the weight of a cycle, we need \( O(n) \)-time.

- Hence, the complexity is: \( \frac{(n-1)!}{2} \cdot O(n) \in O(n \cdot n!) \) (not powerful).
**Time complexity of exhaustive search of TSP**

- Up to shifting, the number of different Hamiltonian cycles on \( n \) vertices is: \( \frac{(n-1)!}{2} \) (use the ”cyclic permutation formula” (https://www.wikiwand.com/en/Cyclic_permutation), and note that the solution set can be grouped into pairs where one is the reflection of the other).

![Diagram of different Hamiltonian cycles](image)

- So, to solve TSP with exhaustive search, then we have to enumerate \( \frac{(n-1)!}{2} \) Hamiltonian cycles, compute their weights, and choose the cycle that has the minimum weight.

- To compute the weight of a cycle, we need \( \mathcal{O}(n) \)-time.

- Hence, the complexity is: \( \frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!) \) (not powerful).

**Exercise.** Given \( n = 20 \), if the time to evaluate one Hamiltonian cycle is 1 second, how much time needed to get the min-weight Hamiltonian cycle!
Up to shifting, the number of different Hamiltonian cycles on \( n \) vertices is: \( \frac{(n-1)!}{2} \) (use the ”cyclic permutation formula” (https://www.wikiwand.com/en/Cyclic_permutation), and note that the solution set can be grouped into pairs where one is the reflection of the other).

So, to solve TSP with exhaustive search, then we have to enumerate \( \frac{(n-1)!}{2} \) Hamiltonian cycles, compute their weights, and choose the cycle that has the minimum weight.

To compute the weight of a cycle, we need \( \mathcal{O}(n) \)-time.

Hence, the complexity is: \( \frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!) \) (not powerful).

Exercise. Given \( n = 20 \), if the time to evaluate one Hamiltonian cycle is 1 second, how much time needed to get the min-weight Hamiltonian cycle?

\[ \approx 1,541,911,905,814 \text{ years} \]
2. The 1/0 knapsack problem (1) [page 143]

Given $n$ items and a knapsack of capacity $K$. Every object $i$ has weight $w_i$ and profit $p_i$. Determine the way to select the objects to the knapsack so that the profit is maximum. The total weight of the objects cannot exceed the knapsack’s capacity.

**Remark.** 1/0 knapsack means that an object can be included to the knapsack (1) or not included (0).
2. The 1/0 knapsack problem (2): algorithm

Exhaustive search for 1/0 knapsack problem:

1. Enumerate all subsets of a set on \( n \) elements
2. Evaluate the profit of every subset in step 1
3. Choose the subset that gives the maximum profit but whose weight does not exceed the knapsack’s capacity
The 1/0 knapsack problem (3): example

Given four objects and a knapsack of capacity $K = 16$. The property of each object is summarized in the following table:

<table>
<thead>
<tr>
<th>Object</th>
<th>Weight</th>
<th>Profit</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>
The optimal solution is given by the subset \( \{2, 3\} \) with profit is 80. So the solution of the problem is given by \( X = \{0, 1, 1, 0\} \) (the objects 1 and 4 are not taken, and objects 2 and 3 are taken).

**Remark.** A solution candidate is "not feasible", because the total weight exceeds the knapsack capacity.
Time complexity:

- The number of subsets of a set of $n$ elements is: $2^n$.
- Time to compute the total weight of a subset is: $O(n)$.
- So, the complexity of exhaustive search for 1/0 knapsack problem is: $O(n \cdot 2^n)$ (exponential complexity).
2. The 1/0 knapsack problem (6): mathematical formulation

We can also express optimization problems mathematically.

Write the solution as \( X = \{x_1, x_2, \ldots, x_n\} \) where:

- \( x_i = 1 \), if the \( i^{th} \) object is selected
- \( x_i = 0 \) otherwise

The mathematical formulation of the 1/0 knapsack problem:

**Definition (math formulation of 1/0 knapsack)**

Maximize \( F = \sum_{i=1}^{n} p_i x_i \)

subject to \( \sum_{i=1}^{n} w_i x_i \leq K \)
and \( x_i = 0 \) or \( x_i = 1 \), for \( i = 1, 2, \ldots, n \)

- Maximize \( F \) : the optimization function
- subject to ... : the constraints (i.e. limitation)
Exhaustive search is used in cryptography as a technique that is used by an attacker to find a decryption key by trying all possible keys, known as exhaustive key search attack or brute force attack.

**Example**

The length of encryption key in DES (Data Encryption Standard) algorithm is 64 bits.

- From that 64 bits, only 56 bits are used, while the other 8 bits are used as parity checking.
- The number of combination for the key is $2^{56} = 72,057,594,037,927,936$
- This means that if the time needed to try one combination is 1 second, then to try all combinations, it would take 2,284,931,317 years.
Given \( n \) staffs and \( n \) tasks. Everyone is assigned to a task. The staff(\( s_i \)) is assigned to the task(\( t_j \)) with cost \( c(i,j) \). Design a brute force algorithm to assign the tasks such that the total cost \( \sum c(i,j) \) is minimized. The instance of the problem is represented in the following matrix.

Example.

Cost matrix:

\[
C = \begin{bmatrix}
\text{task 1} & \text{task 2} & \text{task 3} & \text{task 4} \\
9 & 2 & 7 & 8 \\
6 & 4 & 3 & 7 \\
5 & 8 & 1 & 8 \\
7 & 6 & 9 & 4 \\
\end{bmatrix}
\begin{array}{l}
\text{staff a} \\
\text{staff b} \\
\text{staff c} \\
\text{staff d}
\end{array}
\]
Exercises: 1. Assignment problem (1)

**Remark.** An instance of the assignment problem is completely specified by its cost matrix.

**Question.** How do you view a solution in terms of this matrix?
Remark. An instance of the assignment problem is completely specified by its cost matrix.

Question. How do you view a solution in terms of this matrix?

The first few iterations of solving a small instance of the assignment problem by exhaustive search.

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

\[
\langle 1, 2, 3, 4 \rangle \quad \text{cost} = 9 + 4 + 1 + 4 = 18 \\
\langle 1, 2, 4, 3 \rangle \quad \text{cost} = 9 + 4 + 8 + 9 = 30 \\
\langle 1, 3, 2, 4 \rangle \quad \text{cost} = 9 + 3 + 8 + 4 = 24 \\
\langle 1, 3, 4, 2 \rangle \quad \text{cost} = 9 + 3 + 8 + 6 = 26 \\
\langle 1, 4, 2, 3 \rangle \quad \text{cost} = 9 + 7 + 8 + 9 = 33 \\
\langle 1, 4, 3, 2 \rangle \quad \text{cost} = 9 + 7 + 1 + 6 = 23 \\
\vdots \quad \vdots \\
\]

Remark. \( \langle k, l, m, n \rangle \) means the entries \( c_{1,k}, c_{2,l}, c_{3,m}, c_{4,n} \).
Exercises: 1. Assignment problem (2)

Complexity

- The number of choices: $n \cdot (n - 1) \cdot (n - 2) \cdots 1$
- Complexity: $O(n^2)$
Given \( n \) positive integers. Divide them into two disjoint sets such that the sum of the two subsets are equal. Design an exhaustive search algorithm for this problem.

**Example.** \( n = 6 \), and the integers are 3, 8, 4, 6, 1, 2. It can be divided into \{3, 8, 1\} and \{4, 6, 2\}, where the sum of each of them is 12.

**Question.** How do we obtain this solution?
Algorithm

**Input:** set $S$

**Output:** a subset $A \subseteq S$ satisfying $\text{sum}(A) = \frac{\text{sum}(S)}{2}$

1. Enumerate all possible subsets of $S$;
2. For every subset $A \subseteq S$, check whether $\text{sum}(A) = \frac{\text{sum}(S)}{2}$;

Time complexity: $O(2^n)$ (because a set of $n$ elements has $2^n$ subsets).
Algorithm

**Input:** set $S$

**Output:** a subset $A \subseteq S$ satisfying $\sum(A) = \frac{\sum(S)}{2}$

1. Enumerate all possible subsets of $S$;
2. For every subset $A \subseteq S$, check whether $\sum(A) = \frac{\sum(S)}{2}$;

**Time complexity:** $O(2^n)$
(because a set of $n$ elements has $2^n$ subsets).
A magic square is an arrangement of \( n \) numbers from 1 to \( n^2 \) in a square of size \( n \times n \) such that the sum of every column, row, and diagonal are equal. Design an exhaustive search algorithm to build a magic square of order \( n \).
Algorithm

**Input:** $(1, 2, 3, \ldots n^2)$

**Output:** a magic square of size $n \times n$

1. Enumerate all possible square;
2. For each of them, check if it is a magic square (by checking whether the sum of every row, column, and diagonal is equal).
Exercises: 3. Magic square (part 2)

Algorithm

**Input:** (1, 2, 3, \ldots n^2)

**Output:** a magic square of size $n \times n$

1. Enumerate all possible square;
2. For each of them, check if it is a magic square (by checking whether the sum of every row, column, and diagonal is equal).

**Complexity:** $\mathcal{O}(n!)$ because there are $n!$ possible square
**Task**: Write the pseudocode for those three exercises!
Heuristic is a technique designed for solving a problem more quickly when classic methods are too slow or for finding an approximate solution when classic methods fail to find any exact solution.

- The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem.
- Heuristic uses *guessing*, *intuition*, and *common sense* which cannot be proved mathematically.
- It doesn’t always give an optimal solution.
- A good heuristic can extremely reduce the time to solve a problem by eliminating unnecessary solution candidates.
- No guarantee that heuristic can solve a problem, but it works many times and often faster than exhaustive search.
Heuristic technique can be used to **reduce the number of possible candidates of the problem’s solution**.

**Example**

In **anagram** problem, for English we can use the rule that the letters "c" and "h" often appear consecutively in English words. So we can consider only the permutation of letters where "ch" appear together.

**Example.**

- march → charm
- chapter → patcher, repatch
Example

To solve the magic square problem with exhaustive search, we have to check $9! = 362,880$ possible solution, then check whether for each of them, the sum of every column, row, and diagonal are equal.

With the heuristic technique, for each of the solution, we can check whether the first column has sum $= 15$. If yes, we check the next column/row. Otherwise, we stop, and check the other permutation.
Trade-off: When to choose heuristic technique?

- **Optimality**: When several solutions exist for a given problem, does the heuristic guarantee that the best solution will be found? Is it actually necessary to find the best solution?

- **Completeness**: When several solutions exist for a given problem, can the heuristic find them all? Do we actually need all solutions? Many heuristics are only meant to find one solution.

- **Accuracy and precision**: Can the heuristic provide a confidence interval for the purported solution? Is the error bar on the solution unreasonably large?

- **Execution time**: Is this the best known heuristic for solving this type of problem?