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


Brute force

Part 1: Brute force algorithm (1)

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

Part 1: Brute force algorithm (2)

Example.

- 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

```
procedure MAX(A[1..n])
    max ← a_1
    for i = 2 to n do
        if a_i > max then
            max ← a_i
        end if
    end for
end procedure
```

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

```
procedure MAX(A[1..n])
    max ← a_1
    for i = 2 to n do
        if a_i > max then
            max ← a_i
        end if
    end for
end procedure
```

Complexity? $\mathcal{O}(n)$

2. Sequential search (1)

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.

2. Sequential search (2)

Algorithm 4 Finding an element in an array of integers

procedure SEQSEARCH($A[1..n], x$)
    $i \leftarrow 1$
    while $i < n$ **and** $a_i \neq x$ **do**
        $i \leftarrow i + 1$
    end while
    if $a_i = x$ **then**
        idx $\leftarrow i$
    else
        idx $\leftarrow -1$
    end if
    return idx
end procedure

Complexity? in the assignment!

3. Powering (1)

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

```
procedure POWER(a, n)
    result ← 1
    for i = 1 to n do
        result ← result * a
    end for
    return result
end procedure
```

Algorithm 5 Computing a^n

```
procedure POWER(a, n)
    result ← 1
    for i = 1 to n do
        result ← result * a
    end for
    return result
end procedure
```

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

Is there a better algorithm for "power"?

4. Computing factorial (1)

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

Brute-force approach:

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

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:

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

We multiply the integers 1, 2, until n

Algorithm 8 Computing $n!$

```
procedure FACTORIAL(n)
    result ← 1
    if n <= 1 then return result
    else
        for i = 2 to n do
            result ← result * i
        end for
    end if
    return result
end procedure
```

Algorithm 5 Computing $n!$

```
procedure FACTORIAL(n)
    result ← 1
    if n <= 1 then return result
    else
        for i = 2 to n do
            result ← result * i
        end for
    end if
    return result
end procedure
```

Time complexity: $\mathcal{O}(n)$ (multiplying n numbers)

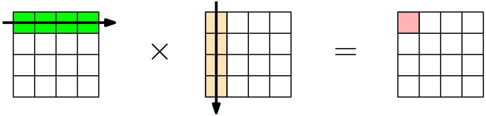
Problem. Given two square matrices, of size $n \times n$.
Find a way to multiply the two matrices!

$$\begin{bmatrix} \blacksquare & \blacksquare & \blacksquare & \cdots & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \cdots & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \cdots & \blacksquare \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \blacksquare & \blacksquare & \blacksquare & \cdots & \blacksquare \end{bmatrix}$$

Figure: A square matrix

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

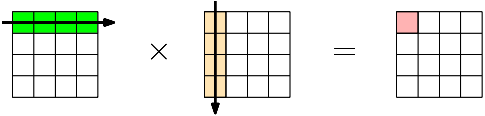
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$



5. Square matrix multiplication (2)

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

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



Brute-force approach: compute each element of C one-by-one by multiplying the corresponding row of A and column of B.

5. Square matrix multiplication (3)

Algorithm 9 Square matrix multiplication

```
procedure MatrixMult(A, B)
    for i ← 1 to n do
        for j ← 1 to n do
            C[i, j] ← 0
            for k ← 1 to n do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
            end for
        end for
    end for
    return C
end procedure
```

5. Square matrix multiplication (4)

Algorithm 9 Square matrix multiplication

```
procedure MatrixMult(A, B)
    for i ← 1 to n do
        for j ← 1 to n do
            C[i, j] ← 0
            for k ← 1 to n do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
            end for
        end for
    end for
    return C
end procedure
```

Time complexity: $\mathcal{O}(n^3)$.

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

6. Prime number test (1)

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.

6. Prime number test (1)

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.

6. Prime number test (2)

Algorithm 10 Prime number test

```
procedure IsPrime(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while isprime & (k ≤ n − 1) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return isprime
    end if
end procedure
```

6. Prime number test (3)

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 muliples 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.

6. Prime number test (4)

Algorithm 11 Prime number test (*sieve of Eratosthenes*)

```
procedure IsPrime2(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while isprime & (k ≤ √n) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return isprime
    end if
end procedure
```

Time complexity:

Algorithm 8 Prime number test

```
procedure IsPrime(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while test & (k ≤ n − 1) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return test
    end if
end procedure
```

Algorithm 9 Prime number test (sieve of Eratosthenes)

```
procedure IsPrime2(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while test & (k ≤ √n) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return test
    end if
end procedure
```

Time complexity:

Algorithm 8 Prime number test

```
procedure IsPrime(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while test & (k ≤ n − 1) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return test
    end if
end procedure
```

Algorithm 9 Prime number test (sieve of Eratosthenes)

```
procedure IsPrime2(n)
    if n < 2 then
        return False
    else
        isprime ← True;   k ← 2
        while test & (k ≤ √n) do
            if n mod k == 0 then
                isprime ← False
            else
                k ← k + 1
            end if
        end while
        return test
    end if
end procedure
```

- Brute-force: $\mathcal{O}(n)$
- Sieve-Erathosthenes brute-force: $\mathcal{O}(\sqrt{n})$

7. Polynomial interpolation (1)

Problem. Evaluate the following polynomial at $x = t$:

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

7. Polynomial interpolation (1)

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

```
procedure POLYNOM(n, A[0..n], t)
    p ← 0
    for i ← n downto 0 do
        power ← 1
        for j ← 1 to i do
            power ← power * t
        end for
        p ← p + a[i] * power
    end for
    return p
end procedure
```

7. Polynomial interpolation (2)

Algorithm 13 Polynomial interpolation

```
procedure POLYNOM(n, A[0..n], t)
    p ← 0
    for i ← n downto 0 do
        power ← 1
        for j ← 1 to i do
            power ← power ∗ t
        end for
        p ← p + a[i] ∗ power
    end for
    return p
end procedure
```

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

Time complexity: $\mathcal{O}(n^2)$.

8. Closest pair problem (1)

Problem. Given n points in the 2 dimensional Eucledian 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 Eucledian 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

```
procedure ClosestPoints(p_1, p_2, ..., p_n)
    dmin ← 999999
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            d ← √((p_i.x − p_j.x)^2 + (p_i.y − p_j.y)^2)
            if d < dmin then
                dmin ← d
                A ← p_i
                B ← p_j
            end if
        end for
    end for
    return A and B
end procedure
```

8. Closest pair problem (3)

Algorithm 13 Finding the closest points

```
procedure ClosestPoints(p_1, p_2, ..., p_n)
    dmin ← 999999
    for i ← 1 to n − 1 do
        for j ← i + 1 to n do
            d ← √((p_i.x − p_j.x)^2 + (p_i.y − p_j.y)^2)
            if d < dmin then
                dmin ← d
                A ← p_i
                B ← p_j
            end if
        end for
    end for
    return A and B
end procedure
```

Time complexity: $\mathcal{O}(n^2)$

9. Pattern matching (1)

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

9. Pattern matching (1)

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.

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
              N O T
```

Example 2.

- 10010101001011110101010001
- 001011

10010101**001011**110101010001 **(Try it!)**

9. Pattern matching (3)

```
procedure PatternMatching(P, T)
    i ← 0;  found ← False
    while (i ≤ n − m) & (not found) do
        j ← 1
        while (j ≤ m) and P_j = T_{i+j} do
            j ← j + 1
        end while
        if j = m then  found ← True
        else i ← i + 1
        end if
    end while
    if found then  return i + 1
    else return −1
    end if
end procedure
```

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 $\rightarrow m$ steps
   - This is done for all possible positions in the string $\rightarrow$ $n - m + 1$ possibilities
   - The number of steps: $m(n = m + 1) \in \mathcal{O}(nm)$
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: $\mathcal{O}(m)$

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.

Characteristic of brute force algorithm (2)

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.

Part 2: Exhaustive search

Exhaustive search is simply a brute-force approach to *combinatorial problems* (permutation, combination, subsets, etc.).

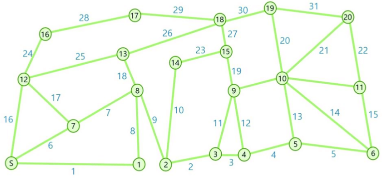
Remark. Example of combinatorial problems are the Traveling Salesman Problem, Knapsack problem, etc.; and non-combinatorial problems are Powering problem, Square Matrix Multipication, etc.

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)!

1. Traveling Salesman Problem (1) [page 142]

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?



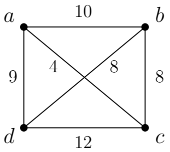
Remark. We may always assume that the given input graph is a **complete graph** (i.e. every pair of vertices is joined by an edge). If as in the figure above the graph is not complete (for example, there is no edge between vertex 1 and 7), then we may assume that the edge $(1, 7)$ exists, but its weight is ∞.

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*.

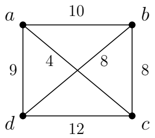
Exhaustive search algorithm for TSP

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.

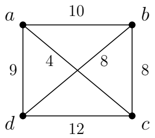
Exercise. Apply the algorithm above to the following graph!



1. Traveling Salesman Problem (3)



No.	Traveling route	Weight
1.	$a \to b \to c \to d \to a$	$10 + 8 + 12 + 9 = 39$
2.	$a \to b \to d \to c \to a$	$10 + 8 + 12 + 4 = 34$
3.	$a \to c \to b \to d \to a$	$4 + 8 + 8 + 9 = 29$
4.	$a \to c \to d \to b \to a$	$4 + 12 + 8 + 10 = 34$
5.	$a \to d \to b \to c \to a$	$9 + 8 + 8 + 4 = 29$
6.	$a \to d \to c \to b \to a$	$9 + 12 + 8 + 10 = 39$



No.	Traveling route	Weight
1.	$a \to b \to c \to d \to a$	$10 + 8 + 12 + 9 = 39$
2.	$a \to b \to d \to c \to a$	$10 + 8 + 12 + 4 = 34$
3.	$a \to c \to b \to d \to a$	$4 + 8 + 8 + 9 = 29$
4.	$a \to c \to d \to b \to a$	$4 + 12 + 8 + 10 = 34$
5.	$a \to d \to b \to c \to a$	$9 + 8 + 8 + 4 = 29$
6.	$a \to d \to c \to b \to a$	$9 + 12 + 8 + 10 = 39$

The shortest route is given by:

- $a \to c \to b \to d \to a$, of weight 29
- $a \to d \to b \to c \to a$, of weight 29

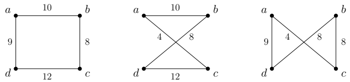
Exhaustive search algorithm for TSP

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

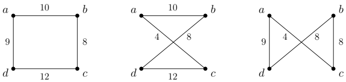
- Up to shifting, the number of different Hamiltonian cycles on n vertices is: $\frac{(n-1)!}{2}$ (use the "cyclic permutation formula" (https://www.wikiwand.com/en/Cyclic_permutation), and note that the solution set can be grouped into pairs where one is the reflection of the other).



- 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 $\mathcal{O}(n)$-time.
- Hence, the complexity is: $\frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!)$ (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).

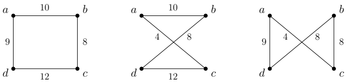


- So, to solve TSP with exhaustive search, then we have to enumerate $\frac{(n-1)!}{2}$ Hamiltonian cycles, compute their weights, and choose the cycle that has the minimum weight.
- To compute the weight of a cycle, we need $\mathcal{O}(n)$-time.
- Hence, the complexity is: $\frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!)$ (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!

Time complexity of exhaustive search of TSP

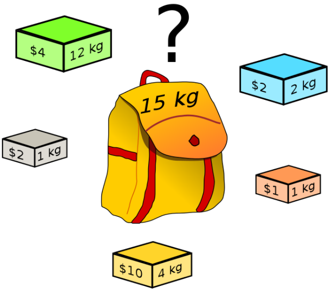
- Up to shifting, the number of different Hamiltonian cycles on n vertices is: $\frac{(n-1)!}{2}$ (use the "cyclic permutation formula" (https://www.wikiwand.com/en/Cyclic_permutation), and note that the solution set can be grouped into pairs where one is the reflection of the other).



- 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 $\mathcal{O}(n)$-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!
$\approx 1,541,911,905,814$ **years**

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 can not exceed the knapsack's capacity.



Remark. 1/0 knapsack means that an object can be included to the knapsack (1) or not included (0).

Exhaustive search for 1/0 knapsack problem:

1. Enumerate all subsets of a set on *n* elements
2. Evaluate the profit of every subset instep 1
3. Choose the subset that gives the maximum profit but whose weight does not exceed the knapsack's capacity

2. 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:

Object	**Weight**	**Profit**
1	2	20
2	5	30
3	10	50
4	5	10

2. The 1/0 knapsack problem (4): example

Subset	Weight	Profit
$\{\}$	0	0
$\{1\}$	2	20
$\{2\}$	5	30
$\{3\}$	10	50
$\{4\}$	5	10
$\{1,2\}$	7	50
$\{1,3\}$	12	70
$\{1,4\}$	7	30

Subset	Weight	Profit
$\{2,3\}$	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

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: $\mathcal{O}(n)$.
- So, the complexity of exhaustive search for 1/0 knapsack problem is: $\mathcal{O}(n \cdot 2^n)$ (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)

$$\textbf{Maximize } F = \sum_{i=1}^{n} p_i x_i$$

$$\textbf{subject to } \sum_{i=1}^{n} w_i x_i \leq K$$

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 in cryptography

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:

$$C = \begin{bmatrix} task\ 1 & task\ 2 & task\ 3 & task\ 4 \\ 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \begin{matrix} staff\ \ a \\ staff\ \ b \\ staff\ \ c \\ staff\ \ d \end{matrix}$$

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.

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

$$\begin{array}{ll} \langle 1,2,3,4 \rangle & \text{cost} = 9+4+1+4 = 18 \\ \langle 1,2,4,3 \rangle & \text{cost} = 9+4+8+9 = 30 \\ \langle 1,3,2,4 \rangle & \text{cost} = 9+3+8+4 = 24 \\ \langle 1,3,4,2 \rangle & \text{cost} = 9+3+8+6 = 26 \\ \langle 1,4,2,3 \rangle & \text{cost} = 9+7+8+9 = 33 \\ \langle 1,4,3,2 \rangle & \text{cost} = 9+7+1+6 = 23 \\ \vdots & \vdots \end{array}$$

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: $n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$
- Complexity: $\mathcal{O}(n^2)$

Exercises: 2. Partition problem (1)

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 $\text{sum}(A) = \frac{\text{sum}(S)}{2}$;

Exercises: 2. Partition problem (1)

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: $\mathcal{O}(2^n)$
(because a set of n elements has 2^n subsets).

Exercises: 3. Magic square (part 1)

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.

4	9	2
3	5	7
8	1	6

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).

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

Exercises: Writing in pseudocode

Task: Write the pseudocode for those three exercises!

Part 3: Heuristic technique (1)

To read:

https://www.wikiwand.com/en/Heuristic_(computer_science)

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.

Part 3: Heuristic technique (2)

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

Part 3: Heuristic technique (3)

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 $\mathbf{sum = 15}$. If yes, we check the next column/row. Otherwise, we stop, and check the other permutation.

Part 3: Heuristic technique (4)

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?