3.1 - Algoritma *Brute Force* (bagian 1)

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 3 (February 2023)

Daftar isi

- Prinsip algoritma brute force
- Beberapa contoh teknik brute-force
  1. Finding max/min of array
  2. Sequential search
  3. Computing power
  4. Menghitung faktorial
  5. Square-matrix multiplication
  6. Prime-number test
  7. Interpolasi polinomial
  8. Masalah pasangan titik terpendek
  9. Pencocokan pola (*pattern matching*)
- Karakteristik algoritma brute force
- *Exhaustive search* (pencarian menyeluruh)
  1. The Traveling Salesman Problem
  2. Permasalahan 1/0 Knapsack
  3. Exhaustive search pada kriptografi
- Latihan: Masalah penugasan; Masalah partisi; *Magic square*
- Teknik heuristik


Brute force

Algoritma brute force (1)

Definisi (Algoritma Brute Force atau *Exhaustive Search*)

Teknik pemecahan masalah yang menggunakan pendekatan langsung.

Solusinya ditemukan dengan memeriksa setiap kemungkinan jawaban satu per satu, untuk menentukan apakah hasilnya merupakan solusi dari permasalahan yang diajukan.

Algoritma brute-force biasanya didasarkan pada:

- pernyataan masalah;
- definisi/konsep yang termuat pada masalah

Karakteristik: sederhana, pendekatan langsung (*direct*), cara yang jelas/terstruktur

Algoritma brute force (2)

Contoh

- Diberikan bilangan bulat n, untuk menemukan semua pembagi dari n, seseorang dapat memeriksa apakah setiap bilangan bulat $i \in [1, n]$ dapat membagi n.

- Diberikan *PIN* yang terdiri dari 4 digit angka (0 samapi dengan 9). Untuk mencari PIN yang benar, brute force akan mencoba semua kemungkinan kombinasi satu per satu seperti 0001, 0002, 0003, 0004, dan seterusnya sampai kita mendapatkan PIN yang tepat (dalam hal ini, paling banyak terdapat 10.000 percobaan).

Bagian 2: Contoh penerapan algoritma *brute force*

1. Menemukan elemen max/min dari sebuah array

Masalah. Diberikan array n bilangan bulat $(a_1, a_2, \ldots, a_n)$. Kita ingin menemukan nilai maksimum pada array.

1. Menemukan elemen max/min dari sebuah array

Masalah. Diberikan array n bilangan bulat $(a_1, a_2, \ldots, a_n)$. Kita ingin menemukan nilai maksimum pada array.

Pendekatan brute-force: untuk menemukan nilai maksimum, bandingkan setiap elemen dari a_1 hingga a_n.

Algorithm 2 Finding maximum of an array of integers

```
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
```

1. Menemukan elemen max/min dari sebuah array

Masalah. Diberikan array n bilangan bulat $(a_1, a_2, \ldots, a_n)$. Kita ingin menemukan nilai maksimum pada array.

Pendekatan brute-force: untuk menemukan nilai maksimum, bandingkan setiap elemen dari a_1 hingga a_n.

Algorithm 3 Finding maximum of an array of integers

```
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
```

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

2. Sequential search (1)

Permasalahan. Diberikan array bilangan bulat $(a_1, a_2, \ldots, a_n)$ dan sebuah bilangan x.

Kita ingin menyelidiki apakah elemen x termuat di dalam array.

- Jika x ditemukan, algoritma menampilkan indeks posisi x dalam array.
- Jika tidak, algoritma memberikan output -1.

Pendekatan brute-force: bandingkan setiap elemen dalam array dengan x. Algoritma berhenti jika x ditemukan atau semua elemen telah diperiksa.

2. Sequential search (2)

Algorithm 4 Finding an element in an array of integers

```
1:  procedure SeqSearch(A[1..n], x)
2:      i ← 1
3:      while i < n and a_i ≠ x do
4:          i ← i + 1
5:      end while
6:      if a_i = x then
7:          idx ← i
8:      else
9:          idx ← −1
10:     end if
11:     return idx
12: end procedure
```

Berapakah kompleksitas algoritma di atas? Kerjakan sebagai latihan!

3. Powering (1)

Permasalahan. Diberikan bilangan riil $a > 0$ dan bilangan bulat tak-n adalah bilangan bukan negatif. Bagaimana menghitung nilai a^n?

3. Powering (1)

Permasalahan. Diberikan bilangan riil $a > 0$ dan bilangan bulat tak-n adalah bilangan bukan negatif. Bagaimana menghitung nilai a^n?

Pendekatan brute-force:

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

Kita mengalikan 1 dengan a sebanyak n kali.

Algorithm 6 Computing a^n

```
1: procedure POWER(a, n)
2:     result ← 1
3:     for i = 1 to n do
4:         result ← result * a
5:     end for
6:     return result
7: end procedure
```

3. Powering (2)

Algorithm 5 Computing a^n

```
1: procedure POWER(a, n)
2:     result ← 1
3:     for i = 1 to n do
4:         result ← result * a
5:     end for
6:     return result
7: end procedure
```

Kompleksitas waktu: $\mathcal{O}(n)$.
Bisakah Anda menjelaskan alasannya?

Apakah ada algoritma yang lebih baik untuk "*Powering* (perpangkatan)"?

4. Menghitung faktorial (1)

Permasalahan. Hitung $n!$ ($n > 0$, n adalah bilangan bulat non-negatif).

Brute-force approach:

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

Kita mengalikan bilangan bulat 1, 2, hingga n

4. Menghitung faktorial (1)

Permasalahan. Hitung $n!$ ($n > 0$, n adalah bilangan bulat non-negatif).

Brute-force approach:

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

Kita mengalikan bilangan bulat 1, 2, hingga n

Algorithm 8 Computing $n!$

```
1:  procedure FACTORIAL(n)
2:      result ← 1
3:      if n <= 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
```

4. Menghitung faktorial (2)

Algorithm 5 Computing $n!$

```
1:  procedure FACTORIAL(n)
2:      result ← 1
3:      if n <= 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
```

Kompleksitas waktu: $\mathcal{O}(n)$ (mengalikan n bilangan)

5. Perkalian matriks persegi (1)

Permasalahan. Diberikan dua matriks persegi, berukuran $n \times n$. Temukan cara untuk mengalikan kedua matriks tersebut!

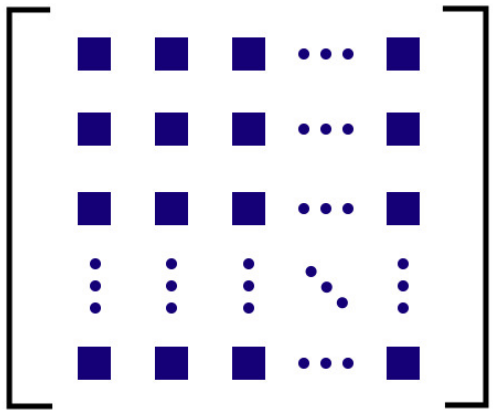
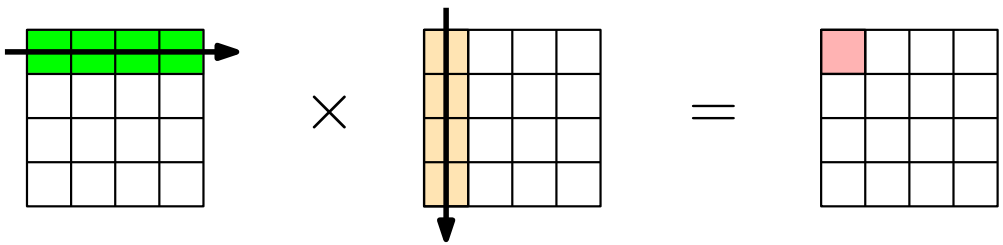


Figure: Sebuah matriks persegi

5. Perkalian matriks persegi (2)

Misalkan $A = [a_{ij}]$, $B = [b_{ij}]$ adalah matriks berukuran $n \times n$, dan $C = A \times B$.

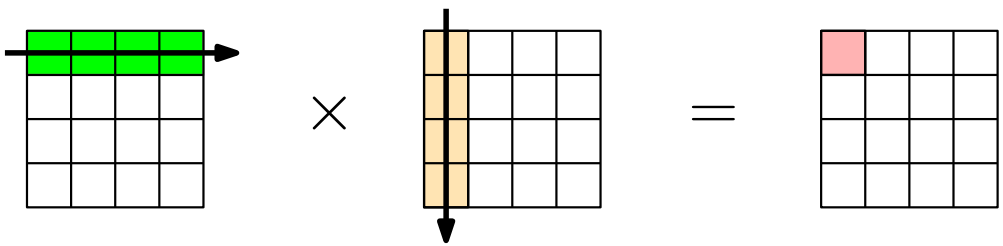
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$



5. Perkalian matriks persegi (2)

Misalkan $A = [a_{ij}]$, $B = [b_{ij}]$ adalah matriks berukuran $n \times n$, dan $C = A \times B$.

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



Pendekatan brute-force: hitung setiap elemen C satu per satu dengan mengalikan baris yang sesuai dari A dan kolom B.

5. Perkalian matriks persegi (3)

Algorithm 9 Perkalian matriks persegi

```
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
6:                  C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
7:              end for
8:          end for
9:      end for
10:     return C
11: end procedure
```

5. Perkalian matriks persegi (4)

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
6:                  C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
7:              end for
8:          end for
9:      end for
10:     return C
11: end procedure
```

Kompleksitas waktu: $\mathcal{O}(n^3)$

Operasi yang "dominan" (memiliki kompleksitas terbesar) dapat dihitung sebagai berikut:

- Loop terdalam memuat: n^3 perkalian, n^3 penjumlahan, dan 3 *assign*
- Loop kedua memuat: n^2 *assign*
- **Return** hanya membutuhkan satu operasi

6. Uji bilangan prima (1)

Permasalahan. Diberi bilangan bulat positif n. Periksa apakah n prima.

Catatan. Bilangan n adalah *prima* jika pembagi n hanyalah 1 dan n.

6. Uji bilangan prima (1)

Permasalahan. Diberi bilangan bulat positif n. Periksa apakah n prima.

Catatan. Bilangan n adalah *prima* jika pembagi n hanyalah 1 dan n.

1. Pendekatan brute-force: bagi n dengan $2, 3, \ldots, n-1$. Jika tidak ada bilangan yang dapat membagi n, maka n adalah bilangan prima.

6. Uji bilangan prima (2)

Algorithm 10 Uji bilangan prima

```
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
```

6. Uji bilangan prima (3)

Permasalahan. Diberi bilangan bulat positif n. Periksa apakah n prima.

Catatan. Bilangan n adalah bilangan prima jika dan hanya jika pembagi dari n hanyalah 1 dan n.

1. Brute-force approach: bagi n dengan $2, 3, \ldots, n-1$. Jika tidak ada yang membagi n, maka n adalah bilangan prima.

2. Sieve of Eratosthenes: untuk batas atas tertentu n, tandai kelipatan bilangan prima secara iteratif sebagai *komposit*. Proses ini berlanjut hingga $p \leq \sqrt{n}$, di mana p adalah bilangan prima.

Dengan teknik ini, untuk memeriksa bahwa n adalah bilangan prima, kita hanya perlu memeriksa apakah ada bilangan prima $\leq \sqrt{n}$ yang membagi n.

6. Uji bilangan prima (4)

Algorithm 11 Uji bilangan prima (*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 ≤ √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 if
15: end procedure
```

6. Uji bilangan prima (5)

Kompleksitas waktu:

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. Uji bilangan prima (5)

Kompleksitas waktu:

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
```

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

7. Interpolasi polinomial (1)

Permasalahan. Tentukan nilai polinomial berikut untuk $x = t$:

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

7. Interpolasi polinomial (1)

Permasalahan. Tentukan nilai polinomial berikut untuk $x = t$:

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

Brute-force approach: masing-masing dari x^k dihitung seperti dalam "algoritma *powering*"; kemudian kalikan x^k dengan a_k, dan jumlahkan dengan suku lainnya.

7. Interpolasi polinomial (2)

Algorithm 12 Interpolasi polinomial

```
1:  procedure POLYNOM(n, A[0..n], t)
2:      p ← 0
3:      for i ← n downto 0 do
4:          power ← 1
5:          for j ← 1 to i do
6:              power ← power ∗ t
7:          end for
8:          p ← p + a[i] ∗ power
9:      end for
10:     return p
11: end procedure
```

7. Interpolasi polinomial (2)

Algorithm 13 Interpolasi polinomial

```
1:  procedure POLYNOM(n, A[0..n], t)
2:      p ← 0
3:      for i ← n downto 0 do
4:          power ← 1
5:          for j ← 1 to i do
6:              power ← power ∗ t
7:          end for
8:          p ← p + a[i] ∗ power
9:      end for
10:     return p
11: end procedure
```

Terdapat $\mathcal{O}(\frac{n(n-1)}{2}) + \mathcal{O}(n+1)$ operasi.

Kompleksitas waktu: $\mathcal{O}(n^2)$.

8. Masalah pasangan titik terpendek (1)

Permasalahan. Diberikan n titik pada ruang Euclid 2 dimensi (dapat dipandang sebagai sistem koordinat Cartesian). Temukan dua titik dengan jarak terpendek.

Jarak antara dua titik $p_1(x_1, y_1)$ dan $p_2 = (x_2, y_2)$ ditentukan dengan formula:

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

8. Masalah pasangan titik terpendek (1)

Permasalahan. Diberikan n titik pada ruang Euclid 2 dimensi (dapat dipandang sebagai sistem koordinat Cartesian). Temukan dua titik dengan jarak terpendek.

Jarak antara dua titik $p_1(x_1, y_1)$ dan $p_2 = (x_2, y_2)$ ditentukan dengan formula:

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

Pendekatan brute-force:

1. Hitung jarak antara setiap pasangan titik
2. Ambil pasangan dengan jarak minimum

8. Masalah pasangan titik terpendek (2)

Algorithm 14 Finding the closest points

```
1:  procedure ClosestPoints(p_1, p_2, ..., p_n)
2:      dmin ← 999999
3:      for i ← 1 to n − 1 do
4:          for j ← i + 1 to n do
5:              d ← sqrt((p_i.x − p_j.x)^2 + (p_i.y − p_j.y)^2)
6:              if d < dmin then
7:                  dmin ← d
8:                  A ← p_i
9:                  B ← p_j
10:             end if
11:         end for
12:     end for
13:     return A and B
14: end procedure
```

8. Masalah pasangan titik terpendek (3)

Algorithm 13 Finding the closest points

```
1:  procedure ClosestPoints(p_1, p_2, ..., p_n)
2:      dmin ← 999999
3:      for i ← 1 to n − 1 do
4:          for j ← i + 1 to n do
5:              d ← sqrt((p_i.x − p_j.x)^2 + (p_i.y − p_j.y)^2)
6:              if d < dmin then
7:                  dmin ← d
8:                  A ← p_i
9:                  B ← p_j
10:             end if
11:         end for
12:     end for
13:     return A and B
14: end procedure
```

Kompleksitas waktu: $\mathcal{O}(n^2)$

9. Pencocokan pola (*pattern matching*) (1)

Permasalahan. Diberikan *sebuah string dengan panjang n* dan *sebuah pola dengan panjang m* dimana $m < n$. Temukan lokasi karakter pertama dari pola pada string yang cocok dengan pola tersebut.

Contoh.

- String: NOBODY NOTICED HIM
- Pattern: NOT

9. Pencocokan pola (*pattern matching*) (1)

Permasalahan. Diberikan *sebuah string dengan panjang n* dan *sebuah pola dengan panjang m* dimana $m < n$. Temukan lokasi karakter pertama dari pola pada string yang cocok dengan pola tersebut.

Contoh.

- String: NOBODY NOTICED HIM
- Pattern: NOT

Brute-force approach:

1. Mulailah dengan karakter pertama dari string.
2. Mulai dari karakter pertama pada pola, periksa apakah pola tersebut cocok dengan suatu *substring*:
   - semua karakter cocok
   - ada karakter yang tidak cocok
3. Jika polanya tidak cocok, kita pindah ke kanan, dan ulangi Langkah 2.

9. Pencocokan pola (*pattern matching*) (2)

Contoh 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
```

Contoh 2.

- 10010101001011110101010001
- 001011

10010101**001011**110101010001 *(Langkah pencocokan dapat dilakukan seperti pada Contoh 1.)*

9. Pencocokan pola (*pattern matching*) (3)

```
1:  procedure PatternMatching(P, T)
2:      i ← 0;  found ← False
3:      while (i ≤ n − m) & (not found) do
4:          j ← 1
5:          while (j ≤ m) and P_j = T_{i+j} do
6:              j ← j + 1
7:          end while
8:          if j = m then  found ← True
9:          else i ← i + 1
10:         end if
11:     end while
12:     if found then  return i + 1
13:     else return −1
14:     end if
15: end procedure
```

9. Pencocokan pola (*pattern matching*) (4)

Kompleksitas waktu

1. Worst case
   - Dalam setiap percobaan pencocokan, kita mencocokkan semua karakter dari pola dengan karakter dalam karakter yang sesuai dari string $\rightarrow$ terdapat m langkah
   - Ini dilakukan untuk semua kemungkinan posisi dalam string $\rightarrow$ terdapat $n - m + 1$ kemungkinan
   - Jumlah langkah: $m(n = m + 1) \in \mathcal{O}(nm)$
2. Best case
   - Ini terjadi ketika pola ditemukan di posisi m pertama dari string
   - Dalam hal ini, kita memeriksa semua karakter pola
   - Kompleksitas: $\mathcal{O}(m)$

Bagian 3: Karakteristik algoritma brute force

Karakteristik algoritma brute force (1)

Keunggulan algoritma brute-force:

- Ini bukan algoritma yang *powerfull*, tetapi hampir semua masalah dapat diselesaikan menggunakan algoritma brute force.
- Sederhana dan mudah dimengerti.
- Dapat diterapkan untuk banyak masalah: pencarian, pengurutan, pencocokan string, perkalian matriks, dll.
- menghasilkan algoritma standar untuk tugas komputasi seperti perkalian/penambahan angka n, menemukan maks/menit dalam larik.

Karakteristik algoritma brute force (2)

Kelemahan algoritma brute-force:

- Algoritmanya tidak "smart", karena membutuhkan banyak perhitungan dan memakan waktu lama untuk memprosesnya. Untuk banyak masalah dunia nyata, banyaknya kandidat biasanya sangat banyak.
- Algoritma brute force cocok untuk instance kecil, karena sederhana dan dapat diimplementasikan dengan mudah.

Catatan. Algoritma ini sering disebut sebagai *algoritma naif*, dan digunakan untuk membandingkan dengan algoritma canggih lainnya.

Bagian 4: *Exhaustive search* (pencarian menyeluruh)

Exhaustive search

Exhaustive search secara sederhana merupakan sebuah pendekatan kasar untuk *masalah kombinatorial* (permutasi, kombinasi, himpunan bagian, dll.).

Catatan. Contoh soal kombinatorial adalah Traveling Salesman Problem, Knapsack problem, dll.; dan masalah non-kombinatorial adalah masalah Powering, Perkalian Matriks Persegi, dll.

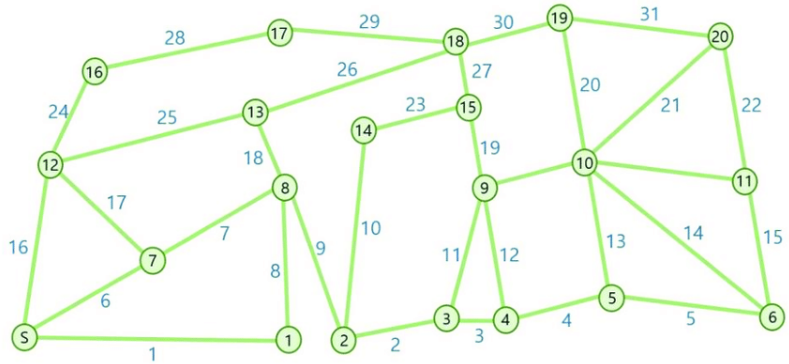
Catatan. Dalam banyak referensi, pencarian lengkap dianggap sama dengan brute force.

Baca buku Anany Levitin, lihat Bagian 3.4 (page 143)!

Bagian 5: Contoh *exhaustive search*

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

Permasalahan. Diberi n kota dan jarak antara setiap pasangan kota, apa rute terpendek yang mengunjungi setiap kota tepat satu kali dan kembali ke kota asal?



Catatan. Kita dapat mengasumsikan bahwa graf input adalah **graf lengkap** (yakni setiap pasangan simpul digabungkan dengan sebuah sisi). Jika graf tidak lengkap (seperti pada gambar di atas, misalnya tidak ada sisi antara simpul 1 dan 7), maka kita dapat mengasumsikan bahwa sisi $(1, 7)$ ada, tetapi bobotnya adalah ∞ (sangat besar).

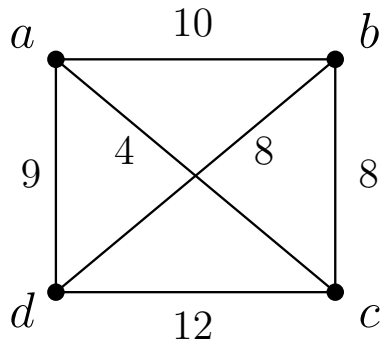
1. Traveling Salesman Problem (2)

Hamiltonian cycle adalah *cycle* (sirkuit) yang mengunjungi setiap simpul pada graf tepat satu kali. Masalah TSP setara dengan *menemukan siklus Hamiltonian dengan bobot minimum*.

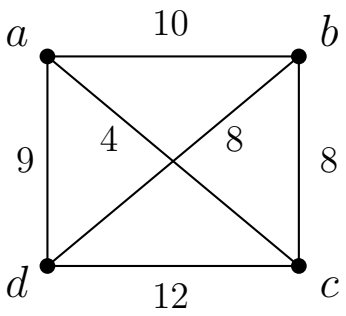
Algoritma pencarian lengkap untuk TSP

1. Menghitung semua siklus Hamiltonian dari graf lengkap n-simpul.
2. Evaluasi bobot setiap siklus Hamiltonian yang ditemukan pada langkah 1.
3. Pilih siklus Hamiltonian dengan bobot minimum.

Latihan. Terapkan algoritma di atas ke grafik berikut!

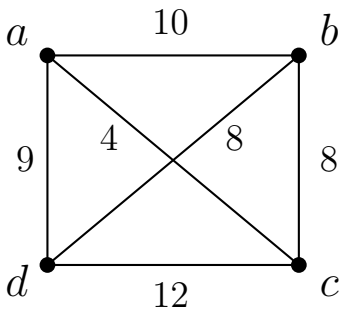


1. Traveling Salesman Problem (3)



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.	$a \to d \to c \to b \to a$	$9 + 12 + 8 + 10 = 39$

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.	$a \to d \to c \to b \to a$	$9 + 12 + 8 + 10 = 39$

Rute terpendek diberikan oleh:

- $a \to c \to b \to d \to a$, dengan bobot 29
- $a \to d \to b \to c \to a$, dengan bobot 29

1. Traveling Salesman Problem (4)

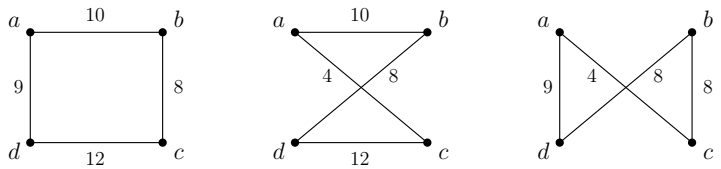
Algoritma exhaustive search untuk solusi TSP

1. Hitung semua siklus Hamiltonian dari graf lengkap n-simpul.
2. Evaluasi bobot setiap siklus Hamiltonian yang ditemukan pada langkah 1.
3. Pilih siklus Hamiltonian dengan bobot minimum.

Diskusikan bagaimana cara menghitung kompleksitasnya?

Kompleksitas waktu of exhaustive search of TSP

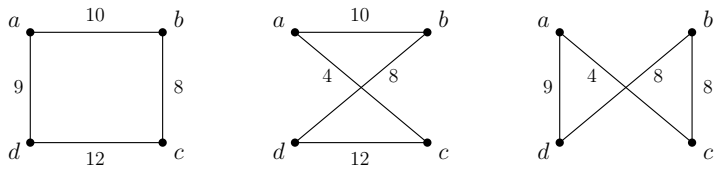
- Up to *shifting*, jumlah siklus Hamiltonian yang berbeda pada n simpul adalah: $\frac{(n-1)!}{2}$ (gunakan "rumus permutasi siklik" (`https://www.wikiwand.com/en/Cyclic_permutation`), dan perhatikan bahwa kumpulan solusi dapat dikelompokkan menjadi pasangan-pasangan di mana yang satu merupakan cerminan dari yang lain).



- Jadi, untuk menyelesaikan TSP dengan pencarian menyeluruh, maka kita harus menghitung $\frac{(n-1)!}{2}$ siklus Hamiltonian, menghitung bobotnya, dan memilih siklus yang memiliki bobot minimum.
- Untuk menghitung bobot dari sebuah siklus, kita memerlukan waktu $\mathcal{O}(n)$.
- Oleh karena itu, kompleksitasnya adalah: $\frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!)$ (tidak cukup baik).

Kompleksitas waktu of exhaustive search of TSP

- Up to *shifting*, jumlah siklus Hamiltonian yang berbeda pada n simpul adalah: $\frac{(n-1)!}{2}$ (gunakan "rumus permutasi siklik" (`https://www.wikiwand.com/en/Cyclic_permutation`), dan perhatikan bahwa kumpulan solusi dapat dikelompokkan menjadi pasangan-pasangan di mana yang satu merupakan cerminan dari yang lain).



- Jadi, untuk menyelesaikan TSP dengan pencarian menyeluruh, maka kita harus menghitung $\frac{(n-1)!}{2}$ siklus Hamiltonian, menghitung bobotnya, dan memilih siklus yang memiliki bobot minimum.
- Untuk menghitung bobot dari sebuah siklus, kita memerlukan waktu $\mathcal{O}(n)$.
- Oleh karena itu, kompleksitasnya adalah: $\frac{(n-1)!}{2} \cdot \mathcal{O}(n) \in \mathcal{O}(n \cdot n!)$ (tidak cukup baik).

Kompleksitas waktu of exhaustive search of TSP

Latihan. Diberikan $n = 20$, jika waktu untuk mengevaluasi satu siklus Hamiltonian adalah 1 detik, berapa banyak waktu yang diperlukan untuk mendapatkan siklus Hamiltonian min-weight!

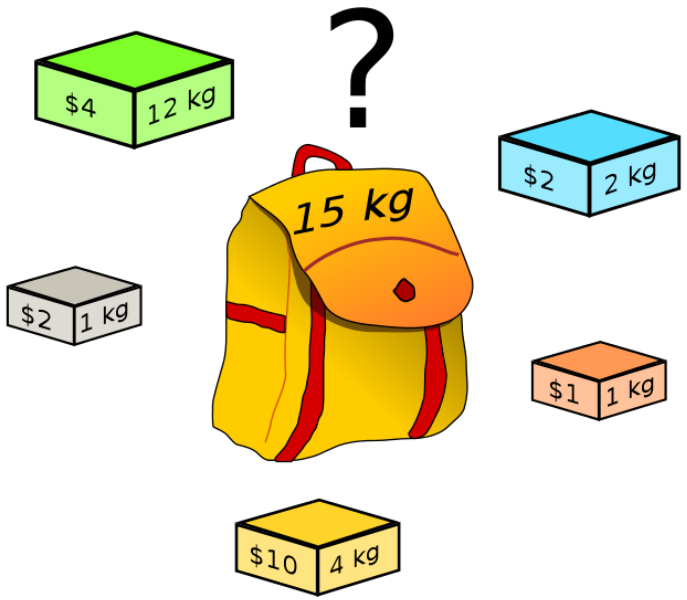
Kompleksitas waktu of exhaustive search of TSP

Latihan. Diberikan $n = 20$, jika waktu untuk mengevaluasi satu siklus Hamiltonian adalah 1 detik, berapa banyak waktu yang diperlukan untuk mendapatkan siklus Hamiltonian min-weight!

$\approx 1,541,911,905,814$ **tahun**

2. Permasalahan 1/0 Knapsack (1) [page 143]

Diberi n item dan ransel berkapasitas K. Setiap objek i memiliki bobot w_i dan untung p_i. Tentukan cara menyeleksi objek ke dalam ransel agar keuntungan maksimal. Berat total benda tidak boleh melebihi kapasitas ransel.



Catatan. 1/0 knapsack artinya suatu objek dapat dimasukkan ke dalam knapsack (1) atau tidak termasuk (0).

2. Permasalahan 1/0 Knapsack (2): algoritma

Exhaustive search untuk 1/0 knapsack problem:

1. Tentukan semua himpunan bagian dari himpunan pada elemen n
2. Evaluasi keuntungan dari setiap subset pada langkah 1
3. Pilih subset yang memberikan keuntungan maksimum tetapi bobotnya tidak melebihi kapasitas ransel

2. Permasalahan 1/0 Knapsack (3): contoh

Diberikan empat objek dan ransel berkapasitas $K = 16$. Karakteristik dari setiap objek dirangkum dalam tabel berikut:

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

2. Permasalahan 1/0 Knapsack (4): contoh

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

Solusi optimal diberikan oleh subset $\{2,3\}$ dengan profit 80. Jadi solusi dari soal tersebut adalah $X = \{0,1,1,0\}$ (objek 1 dan 4 tidak diambil, dan objek 2 dan 3 diambil).

Catatan. Kandidat solusi "tidak layak", karena berat total melebihi kapasitas knapsack.

2. Permasalahan 1/0 Knapsack (5): Analisis kompleksitas waktu

Kompleksitas waktu:

- Jumlah himpunan bagian dari sekumpulan elemen n adalah: 2^n.
- Waktu untuk menghitung bobot total subset adalah: $\mathcal{O}(n)$.
- Jadi, kompleksitas pencarian lengkap untuk 1/0 masalah ransel adalah: $\mathcal{O}(n \cdot 2^n)$ (kompleksitas eksponensial).

2. Permasalahan 1/0 Knapsack (6): formulasi matematis

Kita juga dapat mmerepresentasikan masalah pengoptimalan secara matematis.

Tulis solusinya sebagai $X = \{x_1, x_2, \ldots, x_n\}$ di mana:

- $x_i = 1$, jika objek ke-i dipilih
- $x_i = 0$ sebaliknya

Rumusan matematis Permasalahan 1/0 Knapsack:

Definisi (math formulation of 1/0 knapsack)

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

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

- Maksimalkan F : fungsi pengoptimalan
- tunduk pada ... : kendala (yaitu batasan)

3. Pencarian *exhaustive* pada bidang kriptografi

Exhaustive search digunakan dalam kriptografi sebagai teknik yang digunakan oleh penyerang untuk menemukan kunci dekripsi dengan mencoba semua kunci yang mungkin, dikenal sebagai exhaustive key search attack atau brute force attack.

Example

Panjang kunci enkripsi pada algoritma DES (Data Encryption Standard) adalah 64 bit.

- Dari 64 bit tersebut, hanya 56 bit yang digunakan, sedangkan 8 bit lainnya digunakan sebagai pengecekan paritas.
- Jumlah kombinasi kunci adalah $2^{56} = 72,057,594,037,927,936$
- Artinya jika waktu yang diperlukan untuk mencoba satu kombinasi adalah 1 detik, maka untuk mencoba semua kombinasi membutuhkan waktu 2.284.931.317 tahun.

Bagian 6: Latihan

Latihan: 1. Masalah penugasan (*assignment problem*) (1) [page 145]

Diberikan n staf dan n tugas. Setiap orang diberi tugas. Staff (s_i) ditugaskan ke tugas (t_j) dengan biaya $c(i,j)$. Rancang algoritma brute force untuk menetapkan tugas sedemikian rupa sehingga total biaya $\sum c(i,j)$ diminimalkan. Instance dari masalah direpresentasikan dalam matriks berikut.

Example.

Cost matrix:

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

Latihan: 1. Masalah penugasan (*assignment problem*) (1)

Catatan. Sebuah instance dari masalah penugasan (*masalah penugasan*) secara lengkap ditentukan oleh matriks biayanya.

Question. Bagaimana Anda melihat solusi dalam hal matriks ini?

Latihan: 1. Masalah penugasan (*assignment problem*) (1)

Catatan. Sebuah instance dari masalah penugasan (*masalah penugasan*) secara lengkap ditentukan oleh matriks biayanya.

Question. Bagaimana Anda melihat solusi dalam hal matriks ini?

Beberapa iterasi pertama untuk memecahkan contoh kecil dari masalah penugasan (*assignment problem*) dengan exhaustive search.

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

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

Catatan. $\langle k, l, m, n \rangle$ berarti entri $c_{1,k}, c_{2,l}, c_{3,m}, c_{4,n}$.

Latihan: 1. Masalah penugasan (*assignment problem*) (2)

Kompleksitas

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

Latihan: 2. Masalah bagianisi (1)

Diberikan n bilangan bulat positif. Bagilah mereka menjadi dua himpunan terpisah sedemikian rupa sehingga jumlah kedua subhimpunan itu sama. Rancang algoritma pencarian lengkap untuk masalah ini.

Contoh. $n = 6$, dan bilangan bulatnya adalah $3, 8, 4, 6, 1, 2$. Itu dapat dibagi menjadi $\{3, 8, 1\}$ dan $\{4, 6, 2\}$, di mana jumlah masing-masingnya adalah 12.

Pertanyaan. Bagaimana menyelesaikan permasalahan ini?

Latihan: 2. Masalah bagianisi (1)

Algoritma

Input: himpunan S

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

1. Hitunglah semua himpunan bagian yang mungkin dari S;
2. Untuk setiap subset $A \subseteq S$, periksa apakah $\text{sum}(A) = \frac{\text{sum}(S)}{2}$;

Latihan: 2. Masalah bagianisi (1)

Algoritma

Input: himpunan S

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

1. Hitunglah semua himpunan bagian yang mungkin dari S;
2. Untuk setiap subset $A \subseteq S$, periksa apakah $\text{sum}(A) = \frac{\text{sum}(S)}{2}$;

Kompleksitas waktu: $\mathcal{O}(2^n)$
(karena satu set elemen n memiliki subset 2^n).

Latihan: 3. Persegi ajaib (bagian 1)

Persegi ajaib adalah susunan n angka dari 1 sampai n^2 dalam kuadrat berukuran $n \times n$ sehingga jumlah setiap kolom, baris, dan diagonal adalah sama. Rancang algoritma pencarian lengkap untuk membangun persegi ajaib berukuran n.

4	9	2
3	5	7
8	1	6

Latihan: 3. Persegi ajaib (bagian 2)

Algoritma

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

Output: sebuah persegi ajaib berukuran $n \times n$

1. Hitung semua kuadrat yang mungkin;
2. Untuk masing-masing, periksa apakah itu adalah persegi ajaib (dengan memeriksa apakah jumlah setiap baris, kolom, dan diagonal sama).

Latihan: 3. Persegi ajaib (bagian 2)

Algoritma

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

Output: sebuah persegi ajaib berukuran $n \times n$

1. Hitung semua kuadrat yang mungkin;
2. Untuk masing-masing, periksa apakah itu adalah persegi ajaib (dengan memeriksa apakah jumlah setiap baris, kolom, dan diagonal sama).

Kompleksitas: $\mathcal{O}(n!)$ karena ada $n!$ kemungkinan persegi

Latihan: menuliskan dalam bentuk pseudocode

Tugas: Tulis pseudocode untuk ketiga latihan tersebut!

Bagian 7: Teknik heuristik

Teknik heuristik (1)

Bahan bacaan:

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

Heuristik adalah teknik yang dirancang untuk memecahkan masalah lebih cepat ketika metode klasik terlalu lambat atau untuk menemukan solusi perkiraan ketika metode klasik gagal menemukan solusi eksak.

- Tujuan dari heuristik adalah untuk menghasilkan solusi dalam kerangka waktu yang masuk akal yang cukup baik untuk memecahkan masalah.
- Heuristik menggunakan *menebak*, *intuisi*, dan *akal sehat* yang tidak dapat dibuktikan secara matematis.
- Tidak selalu memberikan solusi optimal.
- Heuristik yang baik dapat sangat mengurangi waktu untuk menyelesaikan masalah dengan menghilangkan kandidat solusi yang tidak perlu.
- Tidak ada jaminan bahwa heuristik dapat memecahkan masalah, tetapi ini bekerja berkali-kali dan seringkali lebih cepat daripada pencarian lengkap.

Teknik heuristik (2)

Teknik heuristik dapat digunakan untuk mengurangi jumlah kandidat yang mungkin dari solusi masalah.

Example

Dalam soal anagram, untuk bahasa Inggris kita dapat menggunakan aturan bahwa huruf "c" dan "h" sering muncul berurutan dalam kata bahasa Inggris. Jadi kita hanya dapat mempertimbangkan permutasi huruf di mana "ch" muncul bersamaan.

Example.

- march $\rightarrow$ charm
- chapter $\rightarrow$ patcher, repatch

Teknik heuristik (3)

Example

Untuk mengatasi masalah *magic square* dengan exhaustive search, kita harus memeriksa $9! = 362,880$ solusi yang mungkin, lalu periksa apakah untuk masing-masing solusi, jumlah setiap kolom, baris, dan diagonal sama.

Dengan Teknik Heuristik, untuk setiap solusi, kita dapat memeriksa apakah kolom pertama memiliki **sum = 15**. Jika ya, kita periksa kolom/baris berikutnya. Jika tidak, kita berhenti, dan memeriksa permutasi lainnya.

Teknik heuristik (4)

Trade-off: Kapan harus menggunakan Teknik Heuristik?

- Optimality: Ketika ada beberapa solusi untuk masalah tertentu, apakah heuristik menjamin bahwa solusi terbaik akan ditemukan? Apakah sebenarnya perlu untuk menemukan solusi terbaik?
- Completeness: Ketika ada beberapa solusi untuk masalah tertentu, dapatkah heuristik menemukan semuanya? Apakah kita benar-benar membutuhkan semua solusi? Banyak heuristik hanya dimaksudkan untuk menemukan satu solusi.
- Accuracy and precision: Bisakah heuristik memberikan interval kepercayaan untuk solusi yang diakui? Apakah bilah kesalahan pada solusi terlalu besar?
- Execution time: Apakah ini heuristik yang paling terkenal untuk menyelesaikan masalah seperti ini?

to be continued...