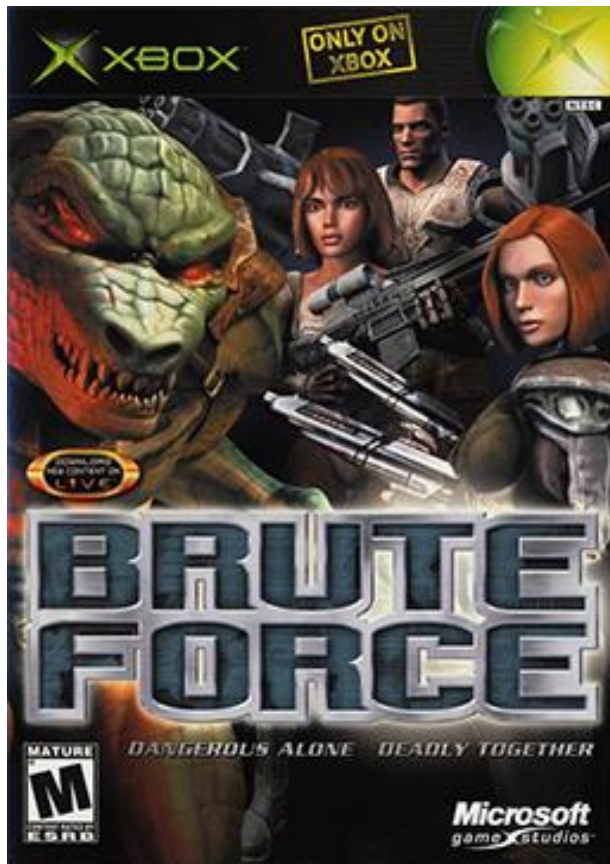


Algoritma *Brute Force*

Oleh: Miftah

Perancangan dan Analisis
Algoritama



?



?

Definisi *Brute Force*

- *Brute force* : pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah
- Biasanya didasarkan pada:
 - pernyataan masalah (*problem statement*)
 - definisi konsep yang dilibatkan.
- Algoritma *brute force* memecahkan masalah dengan
 - sangat sederhana,
 - langsung,
 - jelas (*obvious way*).
- *Just do it!*

Contoh-contoh

(Berdasarkan pernyataan masalah)

1. Mencari elemen terbesar (terkecil)

Persoalan: Diberikan sebuah senarai yang beranggotakan n buah bilangan bulat (a_1, a_2, \dots, a_n). Carilah elemen terbesar di dalam senarai tersebut.

Algoritma *brute force*: bandingkan setiap elemen senarai untuk menemukan elemen terbesar

```

procedure CariElemenTerbesar(input  $a_1, a_2, \dots, a_n$  : integer,
                               output maks : integer)
{ Mencari elemen terbesar di antara elemen  $a_1, a_2, \dots, a_n$ . Elemen
  terbesar akan disimpan di dalam maks.
Masukan:  $a_1, a_2, \dots, a_n$ 
Keluaran: maks
}
Deklarasi
  k : integer

Algoritma:
  maks  $\leftarrow a_1$ 
  for k  $\leftarrow$  2 to n do
    if  $a_k > \text{maks}$  then
      maks  $\leftarrow a_k$ 
    endif
  endfor

```

Kompleksitas waktu algoritma: $O(n)$.

2. *Pencarian beruntun (Sequential Search)*

Persoalan: Diberikan senarai yang berisi n buah bilangan bulat (a_1, a_2, \dots, a_n) . Carilah nilai x di dalam senara tersebut. Jika x ditemukan, maka keluarannya adalah indeks elemen senarai, jika x tidak ditemukan, maka keluarannya adalah 0.

Algoritma *brute force (sequential search)*: setiap elemen senarai dibandingkan dengan x . Pencarian selesai jika x ditemukan atau elemen senarai sudah habis diperiksa.

```

procedure PencarianBeruntun(input  $a_1, a_2, \dots, a_n : \text{integer}$ ,
                              $x : \text{integer}$ ,
                             output  $\text{idx} : \text{integer}$ )
{ Mencari  $x$  di dalam elemen  $a_1, a_2, \dots, a_n$ . Lokasi (indeks elemen)
  tempat  $x$  ditemukan diisi ke dalam  $\text{idx}$ . Jika  $x$  tidak ditemukan, maka
   $\text{idx}$  diisi dengan 0.
  Masukan:  $a_1, a_2, \dots, a_n$ 
  Keluaran:  $\text{idx}$ 
}

Deklarasi
   $k : \text{integer}$ 

Algoritma:
   $k \leftarrow 1$ 
  while ( $k < n$ ) and ( $a_k \neq x$ ) do
     $k \leftarrow k + 1$ 
  endwhile
  {  $k = n$  or  $a_k = x$  }

  if  $a_k = x$  then    {  $x$  ditemukan }
     $\text{idx} \leftarrow k$ 
  else
     $\text{idx} \leftarrow 0$     {  $x$  tidak ditemukan }
  endif

```

Kompleksitas waktu algoritma: $O(n)$.

Adakah algoritma pencarian elemen yang lebih mangkus daripada *brute⁷force*?

Contoh-contoh

(Berdasarkan definisi konsep yang terlibat)

1. **Menghitung a^n ($a > 0$, n adalah bilangan bulat tak-negatif)**

Definisi:

$$\begin{aligned} a^n &= a \times a \times \dots \times a \quad (n \text{ kali}), \text{ jika } n > 0 \\ &= 1 \quad \quad \quad , \text{ jika } n = 0 \end{aligned}$$

Algoritma *brute force*: kalikan 1 dengan a sebanyak n kali


```
function pangkat(a : real, n : integer) → real  
{ Menghitung  $a^n$  }
```

Deklarasi

```
  i : integer  
  hasil : real
```

Algoritma:

```
  hasil ← 1  
  for i ← 1 to n do  
    hasil ← hasil * a  
  end  
  return hasil
```

Kompleksitas waktu algoritma: $O(n)$.

Adakah algoritma perpangkatan yang lebih mangkus daripada *brute force*?

2. Menghitung $n!$ (n bilangan bulat tak-negatif)

Definisi:

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \dots \times n && , \text{ jika } n > 0 \\ &= 1 && , \text{ jika } n = 0 \end{aligned}$$

Algoritma *brute force*: kalikan n buah bilangan, yaitu 1, 2, 3, ..., n , bersama-sama

```
function faktorial(n : integer) → integer
```

```
{ Menghitung n! }
```

Deklarasi

```
  i : integer
```

```
  fak : real
```

Algoritma:

```
  fak ← 1
```

```
  for i ← 1 to n do
```

```
    fak ← fak * i
```

```
  end
```

```
  return fak
```

3. Mengalikan dua buah matriks, A dan B

Definisi:

Misalkan $C = A \times B$ dan elemen-elemen matrik dinyatakan sebagai c_{ij} , a_{ij} , dan b_{ij}

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

- Algoritma *brute force*: hitung setiap elemen hasil perkalian satu per satu, dengan cara mengalikan dua vektor yang panjangnya n .

```

procedure PerkalianMatriks(input A, B : Matriks,
                           input n : integer,
                           output C : Matriks)
{ Mengalikan matriks A dan B yang berukuran  $n \times n$ , menghasilkan
  matriks C yang juga berukuran  $n \times n$ 
  Masukan: matriks integer A dan B, ukuran matriks n
  Keluaran: matriks C
}

```

Deklarasi

```

  i, j, k : integer

```

Algoritma

```

  for i ← 1 to n do
    for j ← 1 to n do
      C[i,j] ← 0      { inisialisasi penjumlah }
      for k ← 1 to n do
        C[i,j] ← C[i,j] + A[i,k]*B[k,j]
      endfor
    endfor
  endfor

```

Adakah algoritma perkalian matriks yang lebih mangkus daripada *brute force*?

- 4. Menemukan semua faktor dari bilangan bulat n (selain dari 1 dan n itu sendiri).**
- Definisi: Bilangan bulat a adalah faktor dari bilangan bulat b jika a habis membagi b .
 - Algoritma *brute force*: bagi n dengan setiap $i = 2, 3, \dots, n - 1$. Jika n habis membagi i , maka i adalah faktor dari n .

```
procedure CariFaktor(input n : integer)  
{ Mencari faktor dari bilangan bulat n selain 1 dan n itu  
sendiri.  
Masukan: n  
Keluaran: setiap bilangan yang menjadi faktor n dicetak.  
}
```

Deklarasi

k : integer

Algoritma:

```
k ← 1  
for k ← 2 to n - 1 do  
  if n mod k = 0 then  
    write(k)  
  endif  
endfor
```

Adakah algoritma pemfaktoran yang lebih baik daripada *brute force*?

5. Tes Bilangan Prima

Persoalan: Diberikan sebuah bilangan bilangan bulat positif. Ujilah apakah bilangan tersebut merupakan bilangan prima atau bukan.

Definisi: bilangan prima adalah bilangan yang hanya habis dibagi oleh 1 dan dirinya sendiri.

Algoritma *brute force*: bagi n dengan 2 sampai \sqrt{n} .
Jika semuanya tidak habis membagi n , maka n adalah bilangan prima.


```

function Prima(input x : integer)→boolean
{ Menguji apakah x bilangan prima atau bukan.
  Masukan: x
  Keluaran: true jika x prima, atau false jika x tidak prima.
}

```

Deklarasi

```

k, y : integer
test : boolean

```

Algoritma:

```

if x < 2 then          { 1 bukan prima }
  return false
else
  if x = 2 then        { 2 adalah prima, kasus khusus }
    return true
  else
    y← $\lceil\sqrt{x}\rceil$ 
    test←true
    while (test) and (y ≥ 2) do
      if x mod y = 0 then
        test←false
      else
        y←y - 1
      endif
    endwhile
    { not test or y < 2 }

    return test
  endif
endif

```

Adakah algoritma pengujian bilangan prima yang lebih mangkus daripada *brute force*?

6. Algoritma Pengurutan *Brute Force*

- Algoritma apa yang memecahkan masalah pengurutan secara *brute force*?

Bubble sort dan *selection sort*!

- Kedua algoritma ini memperlihatkan teknik *brute force* dengan jelas sekali.

Bubble Sort

- Mulai dari elemen ke- n :
 1. Jika $s_n < s_{n-1}$, pertukarkan
 2. Jika $s_{n-1} < s_{n-2}$, pertukarkan
 - ...
 3. Jika $s_2 < s_1$, pertukarkan

→ 1 kali *pass*
- Ulangi lagi untuk pass ke- i , tetapi sampai elemen ke- i
- Semuanya ada $n - 1$ kali *pass*

```
procedure BubbleSort (input/output s : TabelInt, input n : integer)  
{ Mengurutkan tabel s[1..N] sehingga terurut menaik dengan metode  
pengurutan bubble sort.
```

Masukan : Tabel s yang sudah terdefinisi nilai-nilainya.

Keluaran: Tabel s yang terurut menaik sedemikian sehingga
 $s[1] \leq s[2] \leq \dots \leq s[N]$.

```
}
```

Deklarasi

```
  i      : integer      { pencacah untuk jumlah langkah }  
  k      : integer      { pencacah, untuk pengapungan pada setiap  
langkah }  
  temp   : integer      { peubah bantu untuk pertukaran }
```

Algoritma:

```
  for i  $\leftarrow$  1 to n - 1 do  
    for k  $\leftarrow$  n downto i + 1 do  
      if s[k] < s[k-1] then  
        {pertukarkan s[k] dengan s[k-1]}  
        temp  $\leftarrow$  s[k]  
        s[k]  $\leftarrow$  s[k-1]  
        s[k-1]  $\leftarrow$  temp  
      endif  
    endfor  
  endfor
```

Kompleksitas waktu algoritma: $O(n^2)$.

Adakah algoritma pengurutan elemen elemen yang lebih mangkus?

Selection Sort

Pass ke -1 :

1. Cari elemen terbesar mulai di dalam $s[1..n]$
2. Letakkan elemen terbesar pada posisi n (pertukaran)

Pass ke-2:

1. Cari elemen terbesar mulai di dalam $s[1..n - 1]$
2. Letakkan elemen terbesar pada posisi $n - 1$ (pertukaran)

Ulangi sampai hanya tersisa 1 elemen

Semuanya ada $n - 1$ kali *pass*

```

procedure PengurutanSeleksi(input/output  s : array [1..n] of integer)
{ Mengurutkan  $s_1, s_2, \dots, s_n$  sehingga tersusun menaik dengan metode pengurutan seleksi.
  Masukan:  $s_1, s_2, \dots, s_n$ 
  Keluaran:  $s_1, s_2, \dots, s_n$  (terurut menaik)
}

```

Deklarasi

i, j, imaks, temp : integer

Algoritma:

```

for i  $\leftarrow$  n downto 2 do    { jumlah pass sebanyak  $n - 1$  }
  { cari elemen terbesar di dalam  $s[1], s[2], \dots, s[i]$  }
  imaks  $\leftarrow$  1    { elemen pertama diasumsikan sebagai elemen terbesar sementara
  for j  $\leftarrow$  2 to i do
    if  $s[j] > s[\text{imaks}]$  then
      imaks  $\leftarrow$  j
    endif
  endfor
  {pertukarkan  $s[\text{imaks}]$  dengan  $s[i]$  }
  temp  $\leftarrow$   $s[i]$ 
   $s[i] \leftarrow s[\text{imaks}]$ 
   $s[\text{imaks}] \leftarrow$  temp
endfor

```

Kompleksitas waktu algoritma: $O(n^2)$.

Adakah algoritma pengurutan elemen elemen yang lebih mangkus?

7. Mengevaluasi polinom

- Persoalan: Hitung nilai polinom

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

untuk $x = t$.

- Algoritma *brute force*: x^i dihitung secara *brute force* (seperti perhitungan a^n). Kalikan nilai x^i dengan a_i , lalu jumlahkan dengan suku-suku lainnya.

```
function polinom(input t : real)→real  
{ Menghitung nilai p(x) pada x = t. Koefisien-koefisien polinom  
sudah disimpan di dalam a[0..n].  
Masukan: t  
Keluaran: nilai polinom pada x = t.  
}
```

Deklarasi

```
i, j : integer  
p, pangkat : real
```

Algoritma:

```
p ← 0  
for i ← n downto 0 do  
    pangkat ← 1  
    for j ← 1 to i do {hitung xi }  
        pangkat ← pangkat * t  
    endfor  
    p ← p + a[i] * pangkat  
endfor  
return p
```

Kompleksitas algoritma ini adalah $O(n^2)$.

Perbaikan (*improve*):

```
function polinom2(input x0 : real)→real  
{ Menghitung nilai  $p(x)$  pada  $x = t$ . Koefisien-koefisien polinom  
sudah disimpan di dalam  $a[0..n]$ .  
Masukan:  $x_0$   
Keluaran: nilai polinom pada  $x = t$ .  
}
```

Deklarasi

```
i, j : integer  
p, pangkat : real
```

Algoritma:

```
p ← a[n]  
pangkat ← 1  
for i ← 1 to n do  
    pangkat ← pangkat * t  
    p ← p + a[i] * pangkat  
endfor  
return p
```

Kompleksitas algoritma ini adalah $O(n)$.

Adakah algoritma perhitungan nilai polinom yang lebih mangkus daripada *brute force*?

Karakteristik Algoritma *Brute Force*

1. Algoritma *brute force* umumnya tidak “cerdas” dan tidak mangkus, karena ia membutuhkan jumlah komputasi yang besar dalam penyelesaiannya.

Kata “force” mengindikasikan “tenaga” ketimbang “otak”

Kadang-kadang algoritma *brute force* disebut juga algoritma naif (*naïve algorithm*).

2. Algoritma *brute force* lebih cocok untuk masalah yang berukuran kecil.

Pertimbangannya:

- sederhana,
- implementasinya mudah

Algoritma *brute force* sering digunakan sebagai basis pembandingan dengan algoritma yang lebih mangkus.

4. Meskipun bukan metode yang mangkus, hampir semua masalah dapat diselesaikan dengan algoritma *brute force*.

Sukar menunjukkan masalah yang tidak dapat diselesaikan dengan metode *brute force*.

Bahkan, ada masalah yang hanya dapat diselesaikan dengan metode *brute force*.

Contoh: mencari elemen terbesar di dalam senarai.

Contoh lainnya?

- Ken Thompson (salah seorang penemu Unix) mengatakan:

"When in doubt, use brute force",

- Percaya atau tidak, faktanya kernel Unix yang asli lebih menyukai algoritma yang sederhana dan kuat (*robust*) daripada algoritma yang "cerdas" tapi rapuh.

Contoh-contoh lain

1. Pencocokan String (*String Matching*)

Persoalan: Diberikan

- a. teks (*text*), yaitu (*long*) *string* dengan panjang n karakter
- b. *pattern*, yaitu *string* dengan panjang m karakter (asumsi: $m < n$)

Carilah lokasi pertama di dalam teks yang bersesuaian dengan *pattern*.

Algoritma *brute force*:

1. Mula-mula *pattern* dicocokkan pada awal teks.
2. Dengan bergerak dari kiri ke kanan, bandingkan setiap karakter di dalam *pattern* dengan karakter yang bersesuaian di dalam teks sampai:
 - semua karakter yang dibandingkan cocok atau sama (pencarian berhasil), atau
 - dijumpai sebuah ketidakcocokan karakter (pencarian belum berhasil)
3. Bila *pattern* belum ditemukan kecocokannya dan teks belum habis, geser *pattern* satu karakter ke kanan dan ulangi langkah 2.

Contoh 1:

Pattern: NOT

Teks: NOBODY NOTICED HIM

NOBODY **NOT**ICED HIM

1 NOT

2 NOT

3 NOT

4 NOT

5 NOT

6 NOT

7 NOT

8 **NOT**

Contoh 2:

Pattern: 001011

Teks: 10010101**001011**1110101010001

```
    100101010010111110101010001
1  001011
2   001011
3    001011
4     001011
5      001011
6       001011
7        001011
8         001011
9          001011
```

```

procedure PencocokanString(input P : string, T : string,
                           n, m : integer, output idx : integer)
{ Masukan: pattern P yang panjangnya m dan teks T yang
  panjangnya n. Teks T direpresentasikan sebagai string
  (array of character)
Keluaran: lokasi awal kecocokan (idx)
}

```

Deklarasi

```

  i : integer
  ketemu : boolean

```

Algoritma:

```

  i ← 0
  ketemu ← false
  while (i ≤ n-m) and (not ketemu) do
    j ← 1
    while (j ≤ m) and (Pj = Ti+j) do
      j ← j+1
    endwhile
    { j > m or Pj ≠ Ti+j }

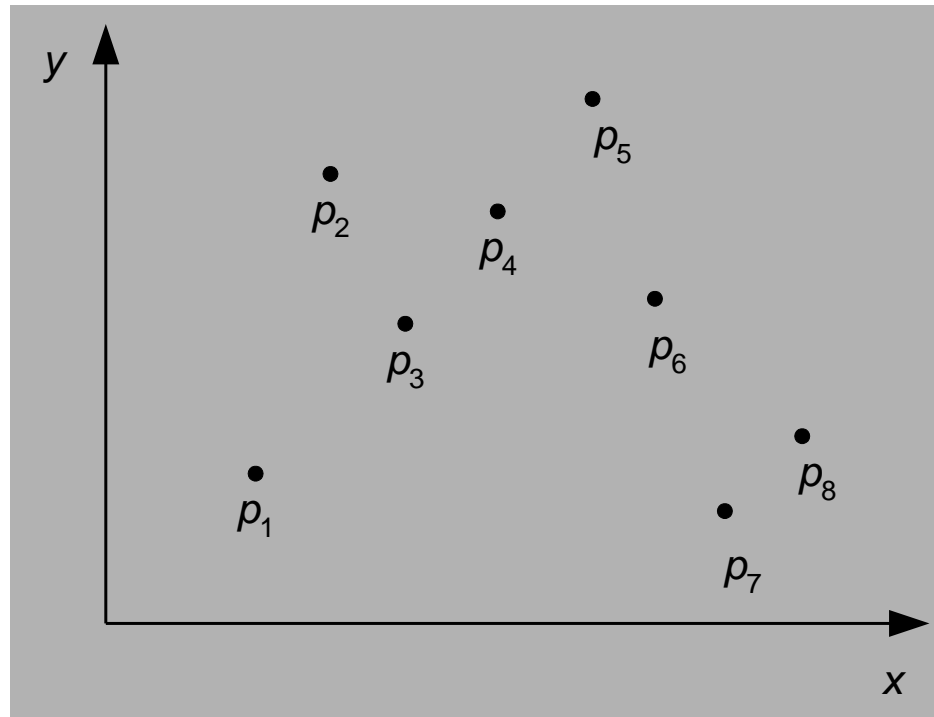
    if j = m then    { kecocokan string ditemukan }
      ketemu ← true
    else
      i ← i+1  {geser pattern satu karakter ke kanan teks }
    endif
  endfor
  { i > n - m or ketemu }
  if ketemu then
    idx ← i+1
  else
    idx ← -1
  endif

```

Kompleksitas algoritma: $O(nm)$ pada kasus terburuk
 $O(n)$ pada kasus rata-rata.

2. Mencari Pasangan Titik yang Jaraknya Terdekat (*Closest Pairs*)

Persoalan: Diberikan n buah titik (2-D atau 3-D), tentukan dua buah titik yang terdekat satu sama lain.



- Jarak dua buah titik, $p_1 = (x_1, y_1)$ dan $p_2 = (x_2, y_2)$ dihitung dengan rumus Euclidean:

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

Algoritma *brute force*:

1. Hitung jarak setiap pasang titik.
 2. Pasangan titik yang mempunyai jarak terpendek itulah jawabannya.
- Algoritma *brute force* akan menghitung sebanyak $C(n, 2) = n(n - 1)/2$ pasangan titik dan memilih pasangan titik yang mempunyai jarak terkecil.

Kompleksitas algoritma adalah $O(n^2)$.

```

procedure CariDuaTitikTerdekat(input P : SetOfPoint,
                                n : integer,
                                output P1, P2 : Point)
{ Mencari dua buah titik di dalam himpunan P yang jaraknya
  terdekat.
  Masukan: P = himpunan titik, dengan struktur data sebagai
  berikut
    type Point = record(x : real, y : real)
    type SetOfPoint = array [1..n] of Point
  Keluaran: dua buah titik, P1 dan P2 yang jaraknya
  terdekat.
}
Deklarasi
  d, dmin : real
  i, j : integer

Algoritma:
  dmin ← 9999
  for i ← 1 to n-1 do

    for j ← i+1 to n do
      d ←  $\sqrt{(P_i.x - P_j.x)^2 + (P_i.y - P_j.y)^2}$ 
      if d < dmin then { perbarui jarak terdekat }
        dmin ← d
        P1 ← Pi
        P2 ← Pj
      endif
    endfor
  endfor

```

Kompleksitas algoritma: $O(n^2)$.

Kekuatan dan Kelemahan Metode *Brute Force*

Kekuatan:

1. Metode *brute force* dapat digunakan untuk memecahkan hampir sebagian besar masalah (*wide applicability*).
2. Metode *brute force* sederhana dan mudah dimengerti.
3. Metode *brute force* menghasilkan algoritma yang layak untuk beberapa masalah penting seperti pencarian, pengurutan, pencocokan *string*, perkalian matriks.
4. Metode *brute force* menghasilkan algoritma baku (standard) untuk tugas-tugas komputasi seperti penjumlahan/perkalian n buah bilangan, menentukan elemen minimum atau maksimum di dalam tabel (*list*).

Kelemahan:

1. Metode *brute force* jarang menghasilkan algoritma yang mangkus.
2. Beberapa algoritma *brute force* lambat sehingga tidak dapat diterima.
3. Tidak sekonstruktif/sekreatif teknik pemecahan masalah lainnya.

Exhaustive Search

Exhaustive search:

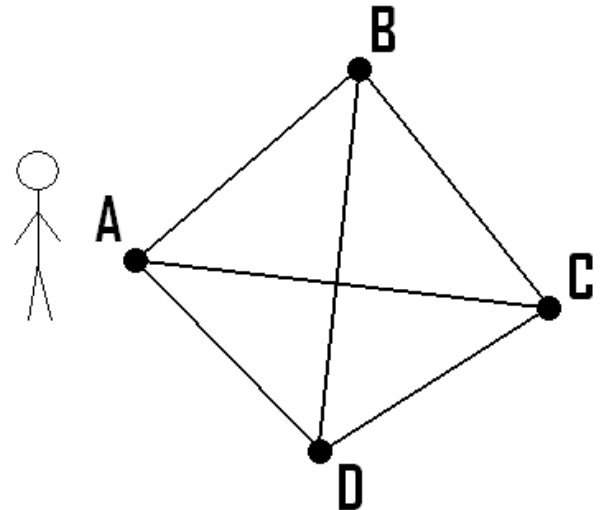
- teknik pencarian solusi secara solusi *brute force* untuk masalah-masalah kombinatorik;
- biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan.

- Langkah-langkah metode *exhaustive search*:
 1. Enumerasi (*list*) setiap solusi yang mungkin dengan cara yang sistematis.
 2. Evaluasi setiap kemungkinan solusi satu per satu, simpan solusi terbaik yang ditemukan sampai sejauh ini (*the best solusi found so far*).
 3. Bila pencarian berakhir, umumkan solusi terbaik (*the winner*)
- Meskipun algoritma *exhaustive* secara teoritis menghasilkan solusi, namun waktu atau sumberdaya yang dibutuhkan dalam pencarian solusinya sangat besar.

Contoh-contoh *exhaustive search*

1. Travelling Salesperson Problem

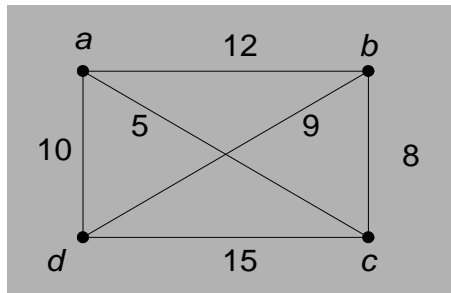
- Persoalan: Diberikan n buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) terpendek yang melalui setiap kota lainnya hanya sekali dan kembali lagi ke kota asal keberangkatan.



- Persoalan *TSP* tidak lain adalah menemukan sirkuit Hamilton dengan bobot minimum.
- Algoritma *exhaustive search* untuk TSP:
 1. Enumerasikan (*list*) semua sirkuit Hamilton dari graf lengkap dengan n buah simpul.
 2. Hitung (evaluasi) bobot setiap sirkuit Hamilton yang ditemukan pada langkah 1.
 3. Pilih sirkuit Hamilton yang mempunyai bobot terkecil.

Contoh 4:

TSP dengan $n = 4$, simpul awal = a



No.	Rute perjalanan (<i>tour</i>)	Bobot
1.	$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$10+12+8+15 = 45$
2.	$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$12+5+9+15 = 41$
3.	$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$10+5+9+8 = \mathbf{32}$
4.	$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$12+5+9+15 = 41$
5.	$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$10+5+9+8 = \mathbf{32}$
6.	$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$10+12+8+15 = 45$

Rute perjalananan terpendek adalah

$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$

$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$

dengan bobot = 32.

- Untuk n buah simpul semua rute perjalanan dibangkitkan dengan permutasi dari $n - 1$ buah simpul.
- Permutasi dari $n - 1$ buah simpul adalah

$$(n - 1)!$$

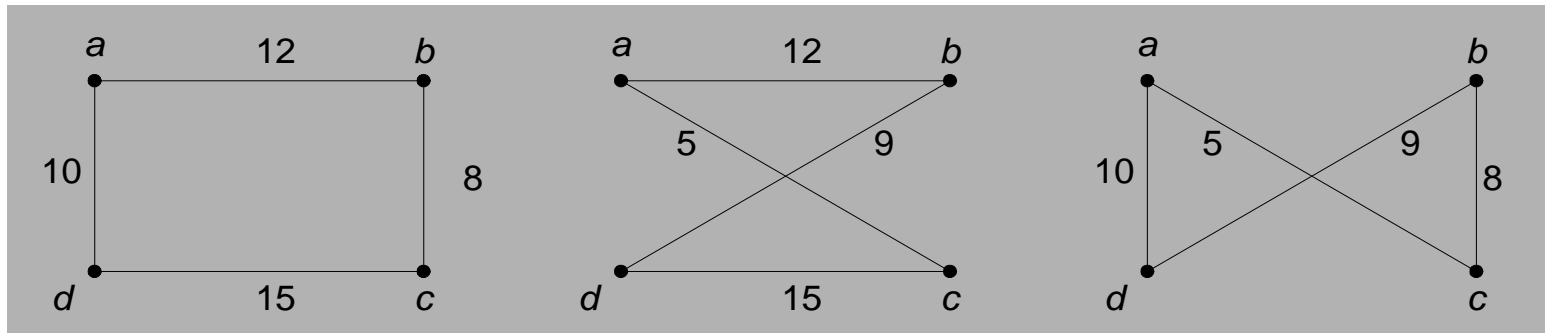
- Pada contoh di atas, untuk $n = 6$ akan terdapat

$$(4 - 1)! = 3! = 6$$

buah rute perjalanan.

- Jika diselesaikan dengan *exhaustive search*, maka kita harus mengenumerasi sebanyak $(n - 1)!$ buah sirkuit Hamilton, menghitung setiap bobotnya, dan memilih sirkuit Hamilton dengan bobot terkecil.
- Kompleksitas waktu algoritma *exhaustive search* untuk persoalan TSP sebanding dengan $(n - 1)!$ dikali dengan waktu untuk menghitung bobot setiap sirkuit Hamilton.
- Menghitung bobot setiap sirkuit Hamilton membutuhkan waktu $O(n)$, sehingga kompleksitas waktu algoritma *exhaustive search* untuk persoalan TSP adalah $O(n \cdot n!)$.

- **Perbaikan:** setengah dari rute perjalanan adalah hasil pencerminan dari setengah rute yang lain, yakni dengan mengubah arah rute perjalanan
 - 1 dan 6
 - 2 dan 4
 - 3 dan 5
- maka dapat dihilangkan setengah dari jumlah permutasi (dari 6 menjadi 3).
- Ketiga buah sirkuit Hamilton yang dihasilkan:



- Untuk graf dengan n buah simpul, kita hanya perlu mengevaluasi $(n - 1)!/2$ sirkuit Hamilton.
- Untuk ukuran masukan yang besar, jelas algoritma *exhaustive search* menjadi sangat tidak mangkus.
- Pada persoalan *TSP*, untuk $n = 20$ akan terdapat $(19!)/2 = 6 \times 10^{16}$ sirkuit Hamilton yang harus dievaluasi satu per satu.

- Sayangnya, untuk persoalan TSP tidak ada algoritma lain yang lebih baik daripada algoritma *exhaustive search*.
- Jika anda dapat menemukan algoritma yang mangkus untuk TSP, anda akan menjadi terkenal dan kaya!
- Algoritma yang mangkus selalu mempunyai kompleksitas waktu dalam orde polinomial.

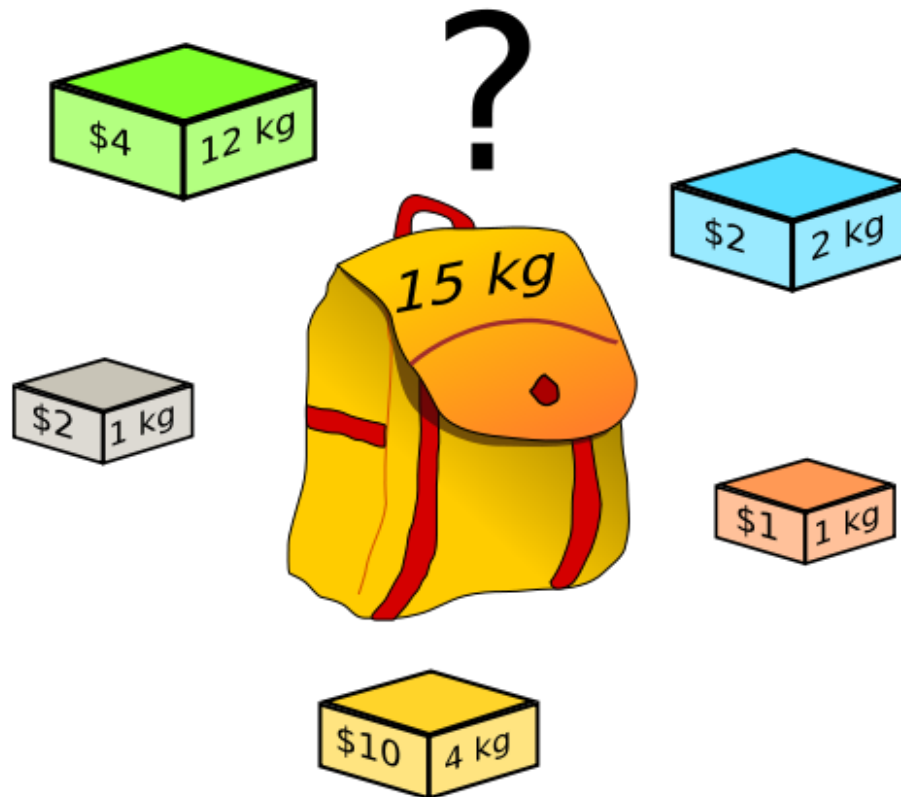
2. $1/0$ Knapsack



- **Persoalan:** Diberikan n buah objek dan sebuah *knapsack* dengan kapasitas bobot K . Setiap objek memiliki properti bobot (*weight*) w_i dan keuntungan(*profit*) p_i .

Bagaimana memilih memilih objek-objek yang dimasukkan ke dalam *knapsack* sedemikian sehingga memaksimalkan keuntungan. Total bobot objek yang dimasukkan ke dalam *knapsack* tidak boleh melebihi kapasitas *knapsack*.

- Persoalan 0/1 *Knapsack* dapat kita pandang sebagai mencari himpunan bagian (*subset*) dari keseluruhan objek yang muat ke dalam *knapsack* dan memberikan total keuntungan terbesar.



- Solusi persoalan dinyatakan sebagai:

$$X = \{x_1, x_2, \dots, x_n\}$$

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

$x_i = 0$, jika objek ke- i tidak dipilih.

Formulasi secara matematis:

$$\text{Maksimasi } F = \sum_{i=1}^n p_i x_i$$

dengan kendala (*constraint*)

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

yang dalam hal ini, $x_i = 0$ atau 1 , $i = 1, 2, \dots, n$

Algoritma *exhaustive search*:

1. Enumerasikan (*list*) semua himpunan bagian dari himpunan dengan n objek.
2. Hitung (evaluasi) total keuntungan dari setiap himpunan bagian dari langkah 1.
3. Pilih himpunan bagian yang memberikan total keuntungan terbesar.

Contoh: $n = 4$.

$$w_1 = 2; \quad p_1 = 20$$

$$w_2 = 5; \quad p_2 = 30$$

$$w_3 = 10; \quad p_3 = 50$$

$$w_4 = 5; \quad p_4 = 10$$

Kapasitas *knapsack* $K = 16$

Langkah-langkah pencarian solusi 0/1
Knapsack secara *exhaustive search*
dirangkum dalam tabel di bawah ini:

Himpunan Bagian	Total Bobot	Total keuntungan
{}	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
{2, 3}	15	80
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	tidak layak
{1, 2, 4}	12	60
{1, 3, 4}	17	tidak layak
{2, 3, 4}	20	tidak layak
{1, 2, 3, 4}	22	tidak layak

- Himpunan bagian objek yang memberikan keuntungan maksimum adalah {2, 3} dengan total keuntungan adalah 80.
- Solusi: $X = \{0, 1, 1, 0\}$

- Berapa banyak himpunan bagian dari sebuah himpunan dengan n elemen? Jawabnya adalah 2^n .

Waktu untuk menghitung total bobot objek yang dipilih = $O(n)$

Sehingga, Kompleksitas algoritma *exhaustive search* untuk persoalan 0/1 *Knapsack* = $O(n \cdot 2^n)$.

- TSP dan 0/1 *Knapsack*, adalah contoh persoalan eksponensial. Keduanya digolongkan sebagai persoalan *NP* (*Non-deterministic Polynomial*), karena tidak mungkin dapat ditemukan algoritma polinomial untuk memecahkannya.

Latihan

(yang diselesaikan secara *exhaustive search*)

1. (**Masalah Penugasan**) Misalkan terdapat n orang dan n buah pekerjaan (*job*). Setiap orang akan di-assign dengan sebuah pekerjaan. Penugasan orang ke- i dengan pekerjaan ke- j membutuhkan biaya sebesar $c(i, j)$. Bagaimana melakukan penugasan sehingga total biaya penugasan adalah seminimal mungkin? Misalkan instansiasi persoalan dinyatakan sebagai matriks C sebagai berikut

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

2. **(Masalah partisi)**. Diberikan n buah bilangan bulat positif. Bagilah menjadi dua himpunan bagian *disjoint* sehingga setiap bagian mempunyai jumlah nilai yang sama (catatan: masalah ini tidak selalu mempunyai solusi).

Contoh: $n = 6$, yaitu 3, 8, 4, 6, 1, 2, dibagi dua menjadi {3, 8, 1} dan {4, 6, 2} yang masing-masing jumlahnya 12.

Rancang algoritma *exhaustive search* untuk masalah ini. Cobalah mengurangi jumlah himpunan bagian yang perlu dibangkitkan.

3. **(Bujursangkar ajaib).** Bujursangkar ajaib (*magic square*) adalah pengaturan n buah bilangan dari 1 hingga n^2 di dalam bujursangkar yang berukuran $n \times n$ sedemikian sehingga jumlah nilai setiap kolom, baris, dan diagonal sama. Rancanglah algoritma *exhaustive search* untuk membangkitkan bujursangkar ajaib orde n .

4	9	2
3	5	7
8	1	6

Exhaustive Search di dalam Kriptografi

- Di dalam kriptografi, *exhaustive search* merupakan teknik yang digunakan penyerang untuk menemukan kunci enkripsi dengan cara mencoba semua kemungkinan kunci.

Serangan semacam ini dikenal dengan nama *exhaustive key search attack* atau *brute force attack*.

- Contoh: Panjang kunci enkripsi pada algoritma *DES (Data Encryption Standard)* = 64 bit.

Dari 64 bit tersebut, hanya 56 bit yang digunakan (8 bit paritas lainnya tidak dipakai).

- Jumlah kombinasi kunci yang harus dievaluasi oleh pihak lawan adalah sebanyak

$$(2)(2)(2)(2)(2) \dots (2)(2) = 2^{56} = 7.205.759.403.7927.936$$

- Jika untuk percobaan dengan satu kunci memerlukan waktu 1 detik, maka untuk jumlah kunci sebanyak itu diperlukan waktu komputasi kurang lebih selama 228.4931.317 tahun!

- Algoritma *exhaustive search* tidak mangkus sebagaimana ciri algoritma *brute force* pada umumnya
- Namun, nilai plusnya terletak pada keberhasilannya yang selalu menemukan solusi (jika diberikan waktu yang cukup).

Mempercepat Algoritma *Exhaustive Search*

- Algoritma *exhaustive search* dapat diperbaiki kinerjanya sehingga tidak perlu melakukan pencarian terhadap semua kemungkinan solusi.
- Salah satu teknik yang digunakan untuk mempercepat pencarian solusi, di mana *exhaustive search* tidak praktis, adalah teknik **heuristik** (*heuristic*).
- Dalam *exhaustive search*, teknik heuristik digunakan untuk mengeliminasi beberapa kemungkinan solusi tanpa harus mengeksplorasinya secara penuh.

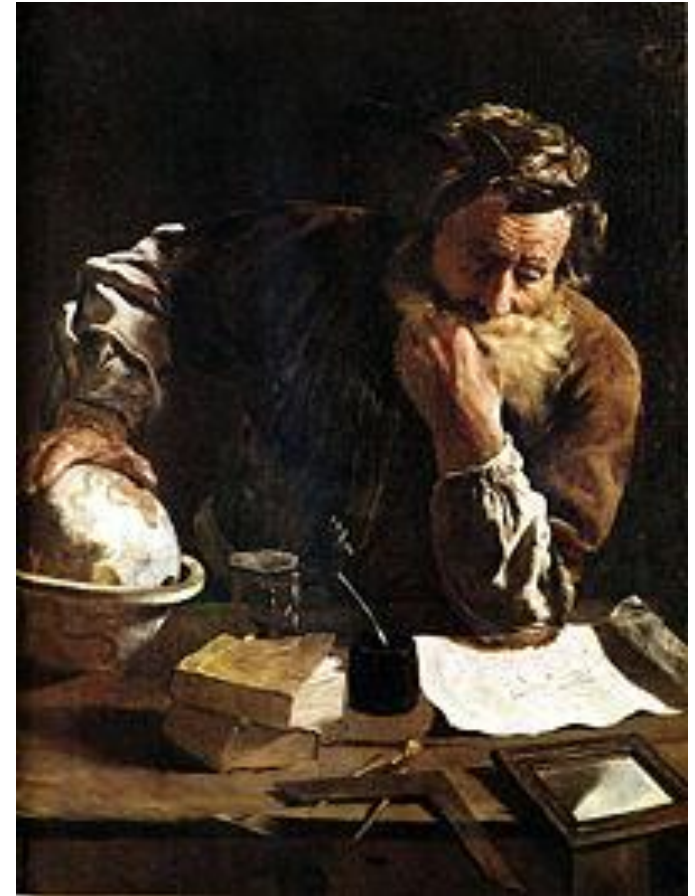
- Heuristik adalah teknik yang dirancang untuk memecahkan persoalan dengan mengabaikan apakah solusi dapat terbukti benar secara matematis
- Contoh dari teknik ini termasuk menggunakan tebakan, penilaian intuitif, atau akal sehat.
- Contoh: program antivirus menggunakan pola-pola heuristik untuk mengidentifikasi dokumen yang terkena virus atau *malware*.

Sejarah

- Heuristik adalah seni dan ilmu menemukan (*art and science of discovery*).

Kata heuristik diturunkan dari Bahasa Yunani yaitu "*eureka*" yang berarti "menemukan" (*to find* atau *to discover*).

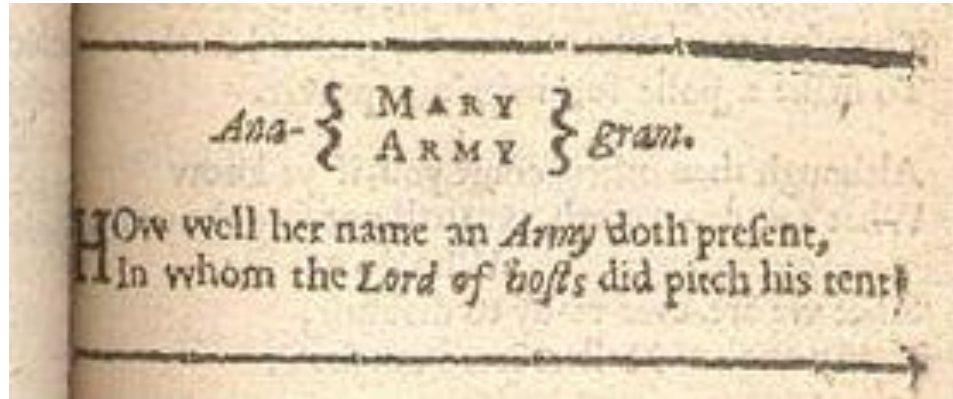
Matematikawan Yunani yang bernama Archimedes yang melontarkan kata "*heureka*", dari sinilah kita menemukan kata "*eureka*" yang berarti "*I have found it.*"



- Heuristik berbeda dari algoritma:
 - heuristik berlaku sebagai panduan (*guideline*),
 - sedangkan algoritma adalah urutan langkah-langkah penyelesaian masalah.
- Heuristik mungkin tidak selalu memberikan hasil yang diinginkan, tetapi secara ekstrim ia berguna pada pemecahan masalah.
- Heuristik yang bagus dapat secara dramatis mengurangi waktu yang dibutuhkan untuk memecahkan masalah dengan cara mengeliminir kebutuhan untuk mempertimbangkan kemungkinan solusi yang tidak perlu.

- Heuristik tidak menjamin selalu dapat memecahkan persoalan, tetapi seringkali memecahkan persoalan dengan cukup baik untuk kebanyakan persoalan, dan seringkali pula lebih cepat daripada pencarian solusi secara *exhaustive search*.
- Sudah sejak lama heuristik digunakan secara intensif di dalam bidang inteligensia buatan (*artificial intelligence*).

- *Contoh penggunaan heuristik untuk mempercepat algoritma exhaustive search*



Contoh: Masalah *anagram*. *Anagram* adalah penukaran huruf dalam sebuah kata atau kalimat sehingga kata atau kalimat yang baru mempunyai arti lain.

Contoh-contoh *anagram* (semua contoh dalam Bahasa Inggris):

lived → *devil*

tea → *eat*

charm → *march*

- Bila diselesaikan secara *exhaustive search*, kita harus mencari semua permutasi huruf-huruf pembentuk kata atau kalimat, lalu memeriksa apakah kata atau kalimat yang terbentuk mengandung arti.
- Teknik heuristik dapat digunakan untuk mengurangi jumlah pencarian solusi. Salah satu teknik heuristik yang digunakan misalnya membuat aturan bahwa dalam Bahasa Inggris huruf *c* dan *h* selalu digunakan berdampingan sebagai *ch* (lihat contoh *charm* dan *march*), sehingga kita hanya membuat permutasi huruf-huruf dengan *c* dan *h* berdampingan. Semua permutasi dengan huruf *c* dan *h* tidak berdampingan ditolak dari pencarian.

- Sumber: Rinaldi Munir: Institut Teknologi Bandung