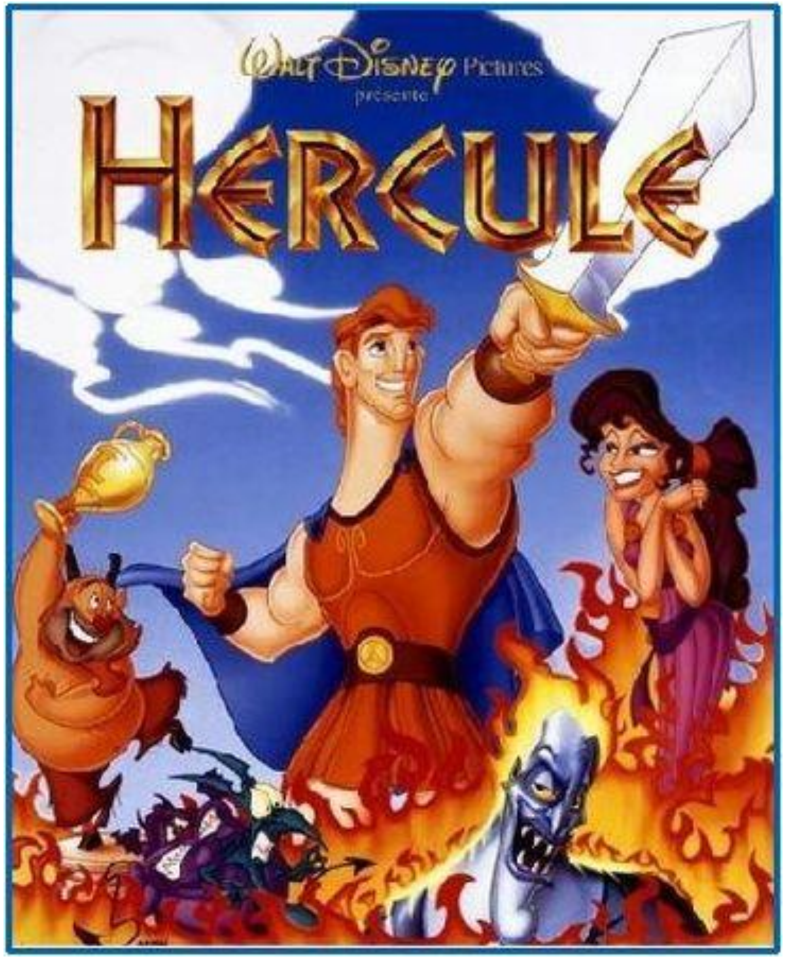
Bahan Kuliah
IF2211 Strategi Algoritma

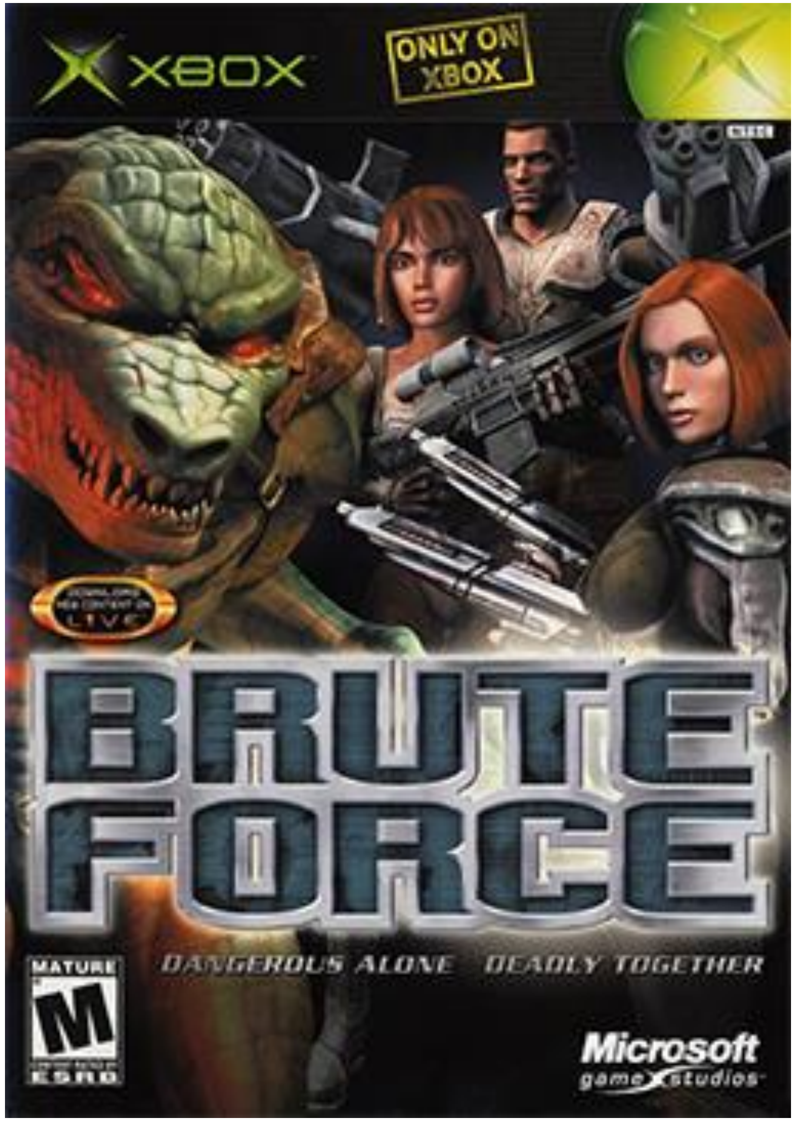
Algoritma *Brute Force*

(Bagian 1)

Oleh: Rinaldi Munir



Program Studi Informatika
Sekolah Teknik Elektro dan Informatika, ITB, 2022


XBOX
ONLY ON XBOX
BRUTE FORCE
DANGEROUS ALONE DEADLY TOGETHER
MATURE
M
ESRB
Microsoft
game studios

Definisi Algoritma *Brute Force*

- Algoritma *Brute force* : pendekatan yang lempang (*straightforward*) untuk memecahkan suatu persoalan

- Biasanya algoritma *brute force* didasarkan pada:
  - pernyataan pada persoalan (*problem statement*)
  - Definisi/konsep yang dilibatkan.

- Algoritma *brute force* memecahkan persoalan dengan
  - sangat sederhana,
  - langsung,
  - jelas caranya (*obvious way*).
  - *Just do it!* atau *Just Solve it!*

Contoh-contoh

(Berdasarkan pernyataan persoalan)

1. **Mencari elemen terbesar (terkecil)**

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

Algoritma *brute force*: bandingkan setiap elemen senarai mulai dari a_1 sampai a_n untuk menemukan elemen terbesar

```
procedure CariElemenTerbesar(input a1, a2, ..., an : integer,  output maks : integer)
{ Mencari elemen terbesar di antara elemen a1, a2, ..., an.
Elemen terbesar disimpan di dalam maks.
Masukan: a1, a2, ..., an
Luaran: maks
}
Deklarasi
  k : integer

Algoritma:
  maks←a1
  for k←2 to n do
      if  ak > maks then
          maks←ak
      endif
  endfor
```

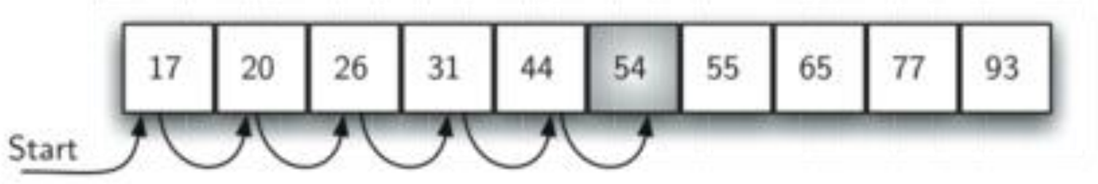
Jumlah operasi perbandingan elemen senarai: n – 1 kali

Kompleksitas waktu algoritma: $O(n)$.

2. *Pencarian beruntun (Sequential Search)*

Persoalan: Diberikan senarai yang berisi n buah bilangan bulat (a_1, a_2, …, a_n). Carilah nilai x di dalam senarai tersebut. Jika x ditemukan, maka luarannya adalah indeks elemen senarai, jika x tidak ditemukan, maka luarannya adalah -1.

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



```
procedure PencarianBeruntun(input a1, a2, ..., an : integer,  x : integer; output idx : integer)
{ Mencari elemen bernilai x di dalam senarai a1, a2, ..., an. Lokasi (indeks elemen) tempat x
ditemukan diisi ke dalam idx. Jika x tidak ditemukan, maka idx diisi dengan 0.
  Masukan: a1, a2, ..., an
  Luaran: idx
}
Deklarasi
  k : integer

Algoritma:
  k←1
  while (k < n) and (ak ≠ x) do
     k ← k + 1
  endwhile
  { k = n or ak = x }

  if  ak = x then   { x ditemukan }
    idx ← k
  else
    idx ←  –1        { x tidak ditemukan }
  endif
```

Jumlah operasi perbandingan elemen senarai maksimal sebanyak: n kali

Kompleksitas waktu algoritma: $O(n)$.

Adakah algoritma pencarian elemen yang lebih mangkus daripada *brute force*?

Contoh-contoh
(Berdasarkan definisi/konsep yang terlibat)

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

Definisi:

$a^n = a \times a \times \ldots \times a$ (n kali) , jika $n > 0$

$= 1$, jika $n = 0$

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
   endfor
   return hasil
```

Jumlah operasi kali: n

Kompleksitas waktu algoritma: $O(n)$.

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

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

Definisi:

$$n! = 1 \times 2 \times 3 \times \ldots \times n \quad , \text{jika } n > 0$$
$$= 1 \quad , \text{jika } n = 0$$

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

```
function faktorial(n : integer) → integer
{ Menghitung n! }

Deklarasi
  k : integer
  fak : real

Algoritma:
    fak ← 1
    for  k ← 1 to n do
        fak ← fak * k
    endfor
    return  fak
```

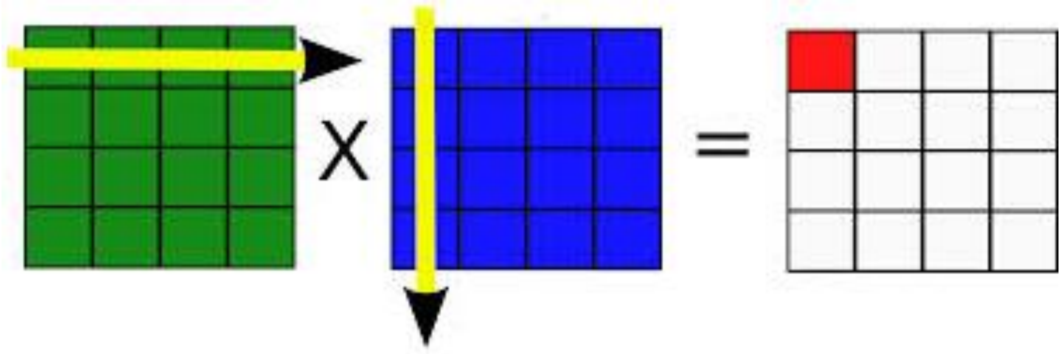
Jumlah operasi kali: *n*
Kompleksitas waktu algoritma: $O(n)$.

3. Mengalikan dua buah matriks, *A* dan *B*, berukuran *n* x *n*

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

Definisi perkalian matriks:

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



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 × n, menghasilkan matriks C yang juga berukuran n × n
  Masukan: matriks integer A dan B, ukuran matriks n
  Luaran: 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
```

Jumlah operasi kali: n^3 dan operasi tambah: n^3, total $2n^3$
Kompleksitas waktu algoritma: $O(n^3)$
Adakah algoritma perkalian matriks yang lebih mangkus daripada *brute force*?

4. Uji Bilangan Prima

Persoalan: Diberikan sebuah bilangan bilangan bulat positif *n*. Ujilah apakah *n* merupakan bilangan prima.

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

Algoritma *brute force*: bagi *n* dengan 2 sampai *n* – 1. Jika semuanya tidak habis membagi *n*, maka *n* adalah bilangan prima.

```
function IsPrima(n : integer) → boolean
{ Menguji apakah n bilangan prima atau bukan.  True jika n prima, atau false jika n tidak prima. }
Deklarasi
    k : integer
    test : boolean

Algoritma:
     test←true
      k ← 2
     while (test) and (k  ≤ n – 1) do
          if  n mod k = 0 then
               test←false
          else
               k ← k + 1
          endif
      endwhile
      { not test or k  = n  }
      return test
   endif
```

Kompleksitas waktu algoritma (kasus terburuk): O(n)

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

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

```
function IsPrima(n : integer) → boolean
{ Menguji apakah n bilangan prima atau bukan.  True jika n prima, atau false jika n tidak prima. }
Deklarasi
   k : integer
   test : boolean

Algoritma:
 if  n < 2 then    { 1 bukan prima }
    return false
 else
    test←true
    k ← 2
   while (test) and (k ≤ √n) do
      if  n mod k = 0 then
          test←false
      else
          k ← k + 1
      endif
    endwhile
    { not test or k > √n }
    return test
  endif
```

Kompleksitas waktu algoritma (kasus terburuk): $O(\sqrt{n})$

Contoh-contoh lainnya

5. Algoritma Pengurutan *Brute Force*

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

Bubble sort dan *selection sort!*

- Kedua algoritma ini memperlihatkan metode *brute force* dengan sangat jelas.

Selection Sort

Pass ke –1:

1. Cari elemen terkecil di dalam *s*[1..*n*]
2. Letakkan elemen terkecil pada posisi ke-1 (lakukan pertukaran)

P*ass* ke-2:

1. Cari elemen terkecil di dalam *s*[2..*n*]
2. Letakkan elemen terkecil pada posisi 2 (pertukaran)

Ulangi sampai hanya tersisa 1 elemen

Semuanya ada *n* –1 kali *pass*

Sumber gambar: **Prof. Amr Goneid**
Department of Computer Science, AUC

```
procedure SelectionSort(input/output  s1, s2, ..., sn : integer)
{ Mengurutkan s1, s2, ..., sn sehingga tersusun menaik dengan metode pengurutan seleksi.
  Masukan: s1, s2, ..., sn
  Luaran: s1, s2, ..., sn (terurut menaik)   }
Deklarasi
    i, j, imin, temp : integer
Algoritma:
    for i ← 1 to n – 1  do    { jumlah pass sebanyak n – 1 }
       { cari elemen terkecil di dalam s[i], s[i+1, ..., s[n] }
       imin ← i  { elemen ke-i diasumsikan sebagai elemen  terkecil sementara }
       for  j ← i+1 to n do
           if  s[j] < s[imin] then
               imin ← j
           endif
       endfor
       {pertukarkan s[imin] dengan s[i] }
       temp ← s[i]
       s[i] ← s[imin]
       s[imin] ← temp
    endfor
```

Jumlah operasi perbandingan elemen larik: $n(n – 1)/2$

Jumlah operasi pertukaran: $n – 1$

Kompleksitas waktu algoritma diukur dari jumlah operasi perbandingan elemen larik: $O(n^2)$.

Adakah algoritma pengurutan yang lebih mangkus daripada *Selection Sort*?

Bubble Sort

- Mulai dari elemen ke-1:

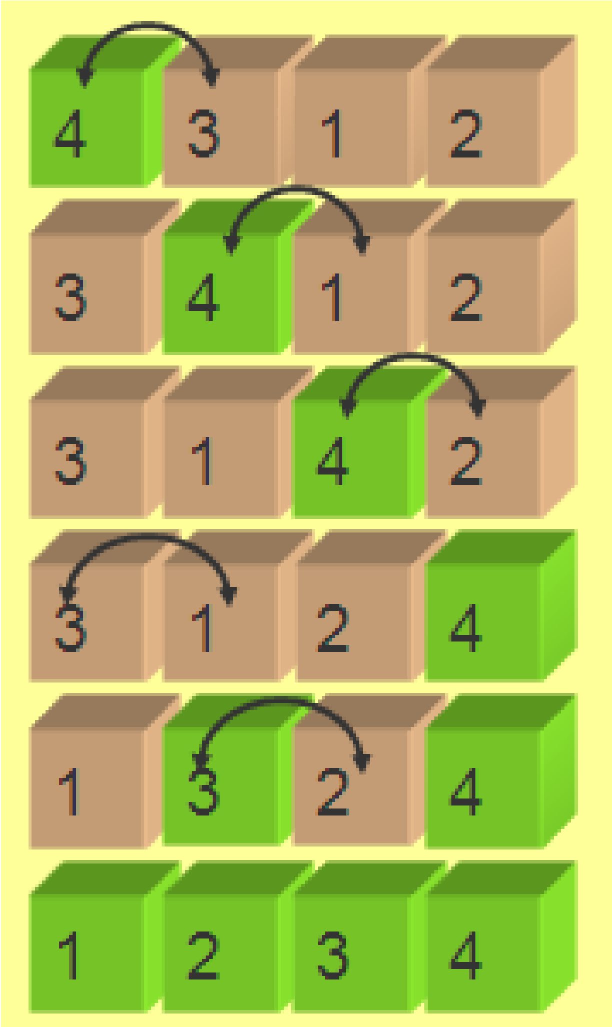
  1. Jika $s_2 < s_1$, pertukarkan
  2. Jika $s_3 < s_2$, pertukarkan

  …

  3. Jika $s_n < s_{n-1}$, pertukarkan

satu kali *pass*

- Ulangi lagi untuk *pass* ke-2, 3, .., n – 1 dst
- Semuanya ada n – 1 kali *pass*



Sumber gambar: **Prof. Amr Goneid**
Department of Computer Science, AUC

```
procedure BubbleSort (input/output s1, s2, ..., sn : integer, input n : integer)
{Mengurutkan s1, s2, ..., sn  sehingga terurut menaik dengan metode pengurutan bubble sort.
 Masukan: s1, s2, ..., sn
 Luaran: s1, s2, ..., sn (terurut menaik)   }
Deklarasi
   i    : integer      { pencacah untuk jumlah langkah }
   k    : integer     { pencacah,untuk pengapungan pada setiap langkah }
   temp : integer  { peubah bantu untuk pertukaran }

 Algoritma:
   for  i ← n – 1  downto 1 do
      for  k ←  1 to i do
          if  s[k+1] < s[k] then
             {pertukarkan s[k] dengan s[k+1]}
             temp ←  s[k]
             s[k] ←  s[k+1]
             s[k+1] ← temp
          endif
      endfor
  endfor
```

Jumlah perbandingan elemen: $n(n-1)/2$
Jumlah pertukaran (kasus terburuk): $n(n-1)/2$
Kompleksitas waktu algoritma diukur dari jumlah perbandingan: $O(n^2)$.
Adakah algoritma pengurutan yang lebih mangkus?

6. Mengevaluasi polinom

- Persoalan: Hitung nilai polinom

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

  pada $x = t$.

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

```
function polinom(t : real)→real
{ Menghitung nilai p(x) pada x = t. Koefisien-koefisein 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 t^i }
           pangkat ← pangkat * t
      endfor
      p ← p + a[i] * pangkat
  endfor
  return  p
```

Jumlah operasi perkalian: $n(n + 1)/2 + (n + 1)$

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

Perbaikan (*improve*): Nilai pangkat pada suku sebelumnya (x^{n-1}) digunakan untuk perhitungan pangkat pada suku sekarang

```
function polinom2(t : real)→real
{ Menghitung nilai p(x) pada x = t. Koefisien-koefisein 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 ← a[0]
 pangkat ← 1
 for  i ← 1 to n  do
          pangkat ← pangkat * t
          p ← p + a[i] * pangkat
 endfor
```

Jumlah operasi perkalian: 2*n*
Kompleksitas algoritma ini adalah *O*(*n*).
Adakah algoritma perhitungan nilai polinom yang lebih mangkus daripada *brute force*?

7. Pencocokan String *(String Matching/Pattern Matching)*

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 cocok (*match*) dengan *pattern*.

Contoh:

Teks: Di mana-mana banyak orang ber**jual**an bakso

Pattern: jual

Algoritma *brute force*:

1. Mula-mula *pattern* disejajarkan (*alignment*) pada awal teks.

2. Dengan menelusuri dari kiri ke kanan pada *pattern*, bandingkan setiap karakter pada *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 kembali langkah 2.

Contoh 1:

Teks: `NOBODY NOTICED HIM`

Pattern: `NOT`

```
  NOBODY NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Contoh 2 (string biner):

Teks: 10010101**001011**110101010001

Pattern: 001011

```
  10010101001011110101010001
1 001011
2  001011
3   001011
4    001011
5     001011
6      001011
7       001011
8        001011
9         001011
```

```
function PencocokanString(input P : string, T : string, m, n : integer, output idx : integer) → integer)
 { Luaran: 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 (P[j] = T[i + j] ) do
         j ← j + 1
     endwhile
     { j > m  or P[j] ≠ T[i+j] }

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

   if   ketemu  then   return  i + 1   else return   –1   endif
```

Brute Force in Java

```
public static int brutematch(String T, String P)
{ int n = T.length();    // n is length of text
  int m = P.length();; // m is length of pattern
  int j;

  for(int i=0; i <= (n-m); i++) {
    j = 0;
    while ((j < m) && (T.charAt(i+j)== P.charAt(j)) ) {
       j++;

    }
    if (j == m)
        return i;   // match at i
  }
  return -1;   // no match
}

 // end of brutematch()
```

Analisis Pencocokan String dengan Brute Force

Worst Case.

- Pada setiap kali pencocokan *pattern*, semua karakter di *pattern* dibandingkan dengan karakter di text pada posisi yang bersesuaian.
- Jadi, setiap kali pencocokan dilakukan m kali perbandingan karakter
- Jumlah pergeseran sampai *pattern* mencapai ujung teks = $(n – m + 1)$
- Total jumlah perbandingan karakter = $m(n – m + 1) = nm – m^2 + m = O(mn)$
- Contoh:
  - T: aaaaaaaaaaaaaaaaaaaaaaaaa
  - P: aaah

    aaah

    ....

    … aaah

Best Case

- *Best case* terjadi bila karakter pertama *pattern P* tidak pernah sama dengan karakter teks *T* yang dicocokkan.
- Jumlah pergeseran *pattern* = (n – m + 1)
- Jumlah perbandingan karakter sebanding dengan jumlah pergeseran *pattern*
- Jumlah perbandingan maksimal n kali:
- Kompleksitas kasus terbaik adalah $O(n)$.

- Contoh:

  T: String ini berakhir dengan zzzz

  P: zzzz

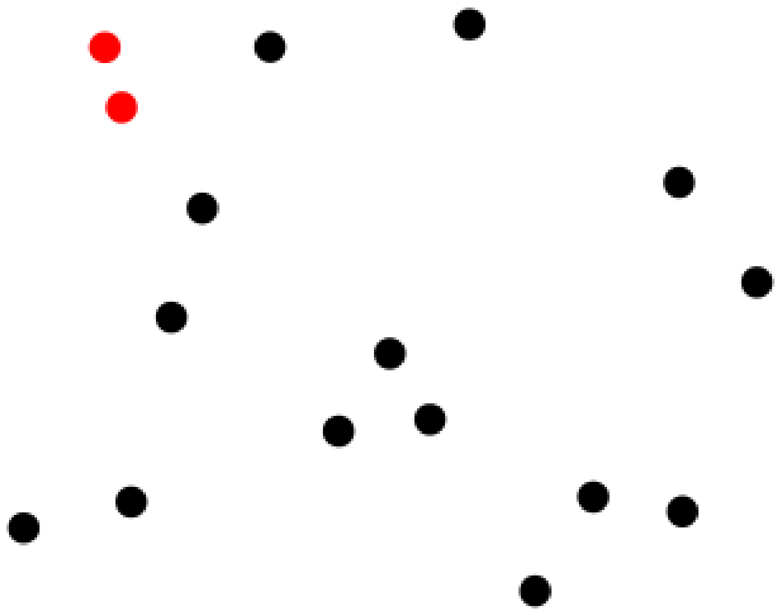
Average Case

- Pencarian pada teks normal (teks biasa)
- Kompleksitas O(m + n)

- Contoh:
  - T: Pada bulan Januari, hujan hampir turun setiap hari
  - P: hujan

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

Persoalan: Diberikan *n* buah titik (pada 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. Untuk setiap titik k, $k = 1, 2, \ldots, n$, hitung jaraknya dengan titik-titik lainnya
  2. Pasangan titik yang mempunyai jarak terpendek itulah jawabannya.

- Ada n buah titik, maka untuk setiap titik dihitung jaraknya dengan n – 1 titik lainnya. Jadi ada n(n – 1) perhitungan jarak dengan rumus Euclidean.

- Untuk setiap titik akan terhitung dua kali dalam perhitungan jarak, jadi sebenarnya hanya terdapat sebanyak $n(n - 1)/2$ perhitungan jarak dengan rumus Euclidean.

- 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
Luaran: 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←√((Pi.x – Pj.x)^2 + ((Pi.y – Pj.y)^2)
          if  d < dmin  then       { perbarui jarak terdekat }
              dmin ← d
              P1 ← Pi
              P2 ← Pj
          endif
      endfor
  endfor
```

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

Karakteristik Algoritma *Brute Force*

1. Algoritma *brute force* umumnya tidak “cerdas” dan tidak mangkus, karena ia membutuhkan volume komputasi yang besar dan waktu yang lama 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 persoalan yang ukuran masukannya (n) kecil.

Pertimbangannya:

- sederhana,
- implementasinya mudah

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

3. Meskipun bukan metode *problem solving* yang mangkus, hampir semua persoalan dapat diselesaikan dengan algoritma *brute force.* Ini adalah kelebihan *brute force*

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

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

Contoh: mencari elemen terbesar di dalam senarai.

Contoh lainnya?

"When in doubt, use brute force" (Ken Thompson, penemu sistem operasi UNIX)

Kekuatan dan Kelemahan Algoritma *Brute Force*

Kekuatan:

1. Algoritma *brute force* dapat diterapkan untuk memecahkan hampir sebagian besar masalah (*wide applicability*).

2. Algoritma *brute force* sederhana dan mudah dimengerti.

3. Algoritma *brute force* menghasilkan algoritma yang layak untuk beberapa masalah penting seperti pencarian, pengurutan, pencocokan *string*, perkalian matriks.

4. Algoritma *brute force* menghasilkan algoritma baku (standard) untuk tugas-tugas komputasi seperti penjumlahan/perkalian *n* buah bilangan, menentukan elemen minimum atau maksimum di dalam senarai (larik).

Kelemahan:

1. Algoritma *brute force* jarang menghasilkan algoritma yang mangkus.

2. Algoritma *brute force* umumnya lambat untuk masukan berukuran besar sehingga tidak dapat diterima.

3. Tidak sekontruktif/sekreatif strategi pemecahan masalah lainnya.

Algoritma *Brute Force* dalam *Sudoku*

- **Sudoku** adalah adalah permainan teka-teki (*puzzle*) logik yang berasal dari Jepang. Permainan ini sangat populer di seluruh dunia.

- Contoh sebuah Sudoku:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- Kotak-kotak di dalam Sudoku harus diisi dengan angka 1 sampai 9 sedemikian sehingga:
    1. tidak ada angka yang sama (berulang) pada setiap baris;
    2. tidak ada angka yang sama (berulang) pada setiap kolom;
    3. tidak ada angka yang sama (berulang) pada setiap bujursangkar (persegi) yang lebih kecil.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

→

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Algoritma *Brute Force* untuk Sudoku:

1. Tempatkan angka “1” pada sel kosong pertama. Periksa apakah penempatan “1” dibolehkan (dengan memeriksa baris, kolom, dan kotak).

2. Jika tidak ada pelanggaran, maju ke sel berikutnya. Tempatkan “1” pada sel tersebut dan periksa apakah ada pelanggaran.

3. Jika pada pemeriksaan ditemukan pelanggaran, yaitu penempatan “1” tidak dibolehkan, maka coba dengan menempatkan “2”.

4. Jika pada proses penempatan ditemukan bahwa tidak satupun dari 9 angka diperbolehkan, maka tinggalkan sel tersebut dalam keadaan kosong, lalu mundur satu langkah ke sel sebelumnya. Nilai di sel tersebut dinaikkan 1.

5. Ulangi Langkah 1 sampai 81 sel sudah terisi solusi yang benar.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Contoh-contoh persoalan *brute force* lainnya

1. Cryptarithmetic

Diberikan sebuah penjumlahan huruf, carilah angka yang merepresentasikan huruf-huruf tersebut.

Contoh:

$$\begin{array}{r} \text{S E N D} \\ +\ \ \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

Solusinya:

$$\begin{array}{r} 9\ 5\ 6\ 7 \\ +\ \ 1\ 0\ 8\ 5 \\ \hline 1\ 0\ 6\ 5\ 2 \end{array}$$

Jadi, S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2

$$\begin{array}{r} \text{J U N E} \\ +\ \text{J U L Y} \\ \hline \text{A P R I L} \end{array} \qquad \begin{array}{r} 7924 \\ +\ 7906 \\ \hline 15830 \end{array}$$

$$\begin{array}{r} \text{F O R T Y} \\ \text{T E N} \\ +\ \ \text{T E N} \\ \hline \text{S I X T Y} \end{array} \qquad \begin{array}{r} 2\ 9\ 7\ 8\ 6 \\ 8\ 5\ 0 \\ +\ \ 8\ 5\ 0 \\ \hline 3\ 1\ 4\ 8\ 6 \end{array}$$

2. Permainan 24 (24 Game)

Diberikan 4 buah bilangan bulat. Bagaimana mengkombinasikan keempat bilangan bulat tersebut dengan operator aritmetika sehingga hasilnya = 24

Welcome to 24 Game Solver

Enter your 4 numbers below, then click "Solve" to see every solution that equals 24.

4 | 7 | 8 | 28 | Solve | Clear

36 solutions found | Highlight similar solutions

(4 * (7 - 8)) + 28	(7 * 8) - (4 + 28)	8 + ((4 / 7) * 28)	((28 * 4) / 7) + 8
(4 / (7 - 8)) + 28	((7 * 8) - 4) - 28	8 + ((4 * 28) / 7)	(28 * (4 / 7)) + 8
((4 / 7) * 28) + 8	((7 - 8) * 4) + 28	(8 * 7) - (4 + 28)	(28 - 4) * (8 - 7)
(4 / (7 / 28)) + 8	(7 - 8) * (4 - 28)	((8 * 7) - 4) - 28	(28 - 4) / (8 - 7)
((4 * 28) / 7) + 8	(7 * 8) - (28 + 4)	(8 * 7) - (28 + 4)	((28 * 7) - 4) / 8
(4 * (28 / 7)) + 8	((7 * 8) - 28) - 4	((8 * 7) - 28) - 4	((28 / 7) * 4) + 8
(4 - 28) * (7 - 8)	((7 * 28) - 4) / 8	((8 - 7) * 28) - 4	(28 / (7 / 4)) + 8
(4 - 28) / (7 - 8)		(8 - 7) * (28 - 4)	28 + ((7 - 8) * 4)
		8 + ((28 * 4) / 7)	(28 * (8 - 7)) - 4
		8 + ((28 / 7) * 4)	(28 / (8 - 7)) - 4
			28 - ((8 - 7) * 4)

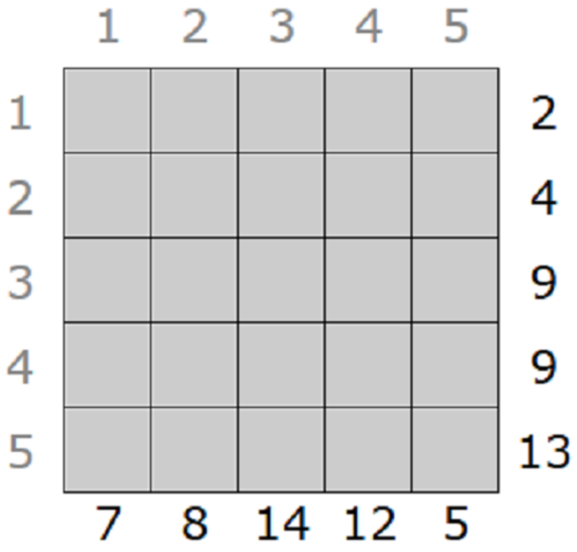
3. Crossword Puzzle

Crossword puzzle atau permainan teka-teki silang adalah pemainan yang memasangkan semua kata yang tersedia ke dalam kotak-kotak yang bersesuaian, baik secara mendatar maupun menurun.

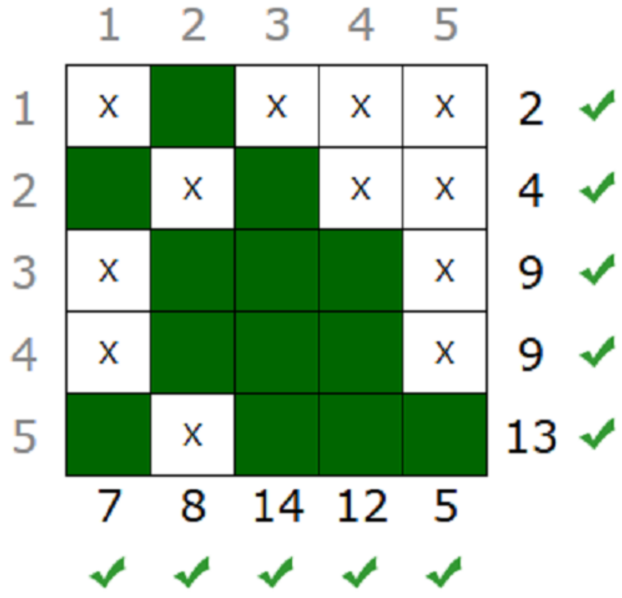
S	T	A	N	D	G	U	A	R	D		B	U	F	F
A		T		I		N		I				K		L
R	U	I	N	S		R	A	S	P	B	E	R	R	Y
G		S		I		E		K		I		A		A
A	R	S	O	N	I	S	T		P	L	A	I	T	S
S		U		T		E		O		L		N		H
S	C	E	N	E		R	E	P	A	I	R	E	D	
O				G		V		H		A				I
	E	M	E	R	G	E	N	T		R	O	B	I	N
L		A		A		D		H		D		L		U
I	N	G	O	T	S		C	A	S	T	D	O	W	N
M		N		E		S		L		A		S		I
B	O	A	R	D	R	O	O	M		B	U	S	T	S
E		T				D		I		L		O		O
R	H	E	A		H	A	T	C	H	E	T	M	A	N

4. Kakurasu

Kakurasu adalah permainan teka-teki logika yang berasal dari Jepang. Teka-teki dimainkan pada grid persegi, misalnya 4 x 4, 5 x 5, 6 x 6, dst. Pada setiap pinggir kotak paling kanan dan kotak paling bawah terdapat angka-angka. Tujuan permainannya adalah memilih kotak-kotak di dalam grid sehingga jumlah nilainya sama dengan angka yang ditunjukkan di kotak paling kanan dan kotak paling bawah (secara horizontal dan vertikal).



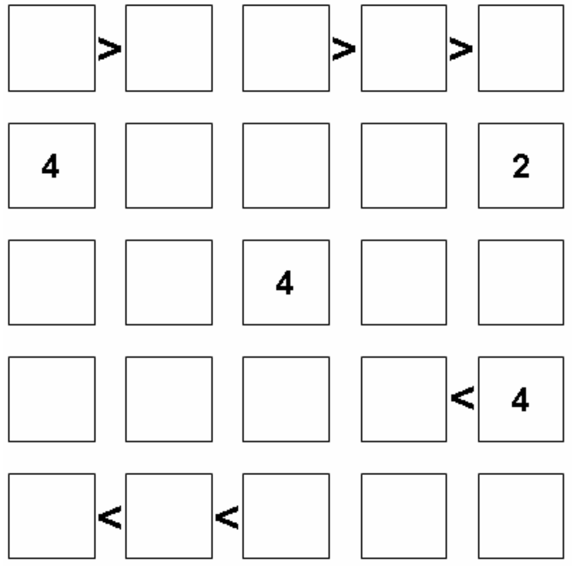
Keadaan awal



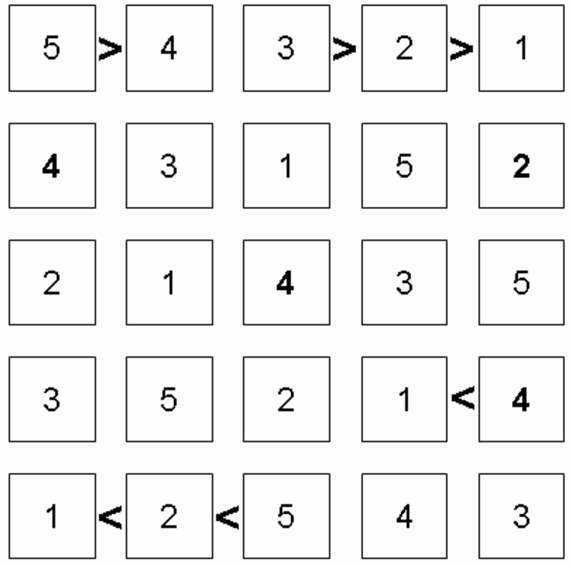
Solusi

5. Futoshiki

Fotoshiki (不等式) adalah adalah permainan teka-teki logika yang berasal dari Jepang. Teka-teki dimainkan pada grid persegi, misalnya 5 x 5 . Tujuannya adalah untuk menempatkan angka 1 sampai 5 (atau dimensi lainnya) sehingga setiap baris dan kolom berisi masing-masing angka 1 sampai 5 . Beberapa angka diberikan di awal sebagai panduan. Selain itu, beberapa tanda ketidaksamaan (< atau >) diberikan di antara beberapa kotak, sedemikian sehingga nilai yang satu harus lebih lebih tinggi atau lebih rendah dari tetangganya (Sumber: Wikipedia)



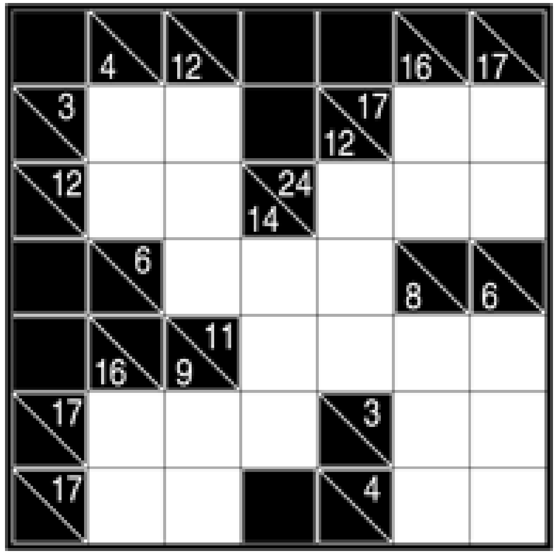
Keadaan Awal Futoshiki



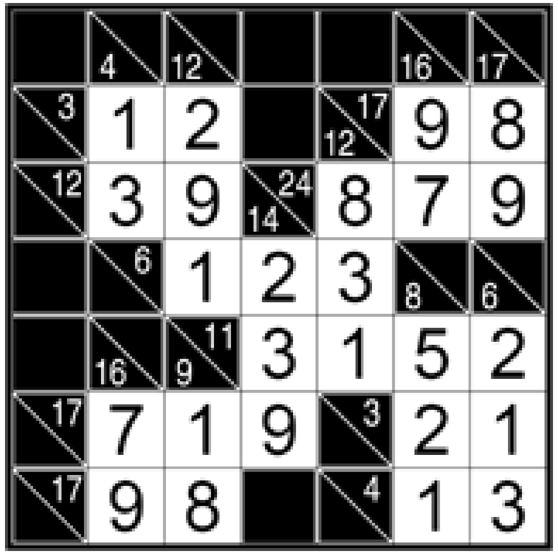
Solusi

6. Kakuro

Tujuan permainan adalah untuk mengisi semua kotak kosong di dalam *grid* dengan hanya 1-9 angka sehingga angka-angka yang anda masukkan jumlahnya sama dengan *clue* (penunjuk) yang terletak pada *grid* di sebelah kiri dan *grid* sebelah. Jumlah dari setiap blok horisontal sama dengan petunjuk di sebelah kiri, dan jumlah dari setiap blok vertikal sama dengan petunjuk di atasnya. Selain itu, tidak boleh yang sama digunakan di dalam blok yang sama lebih dari sekali



Keadaan Awal Kakuro



Solusi

Exhaustive Search

Exhaustive search:

- adalah teknik pencarian solusi secara solusi *brute force* untuk persoalan-persoalan kombinatorik;

- yaitu persoalan di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan.

- Langkah-langkah di dalam *exhaustive search*:

  1. Enumerasi (*list*) setiap kemungkinan solusi dengan cara yang sistematis.

  2. Evaluasi setiap kemungkinan solusi satu per satu, simpan solusi terbaik yang ditemukan sampai sejauh ini (*the best solution found so far*).

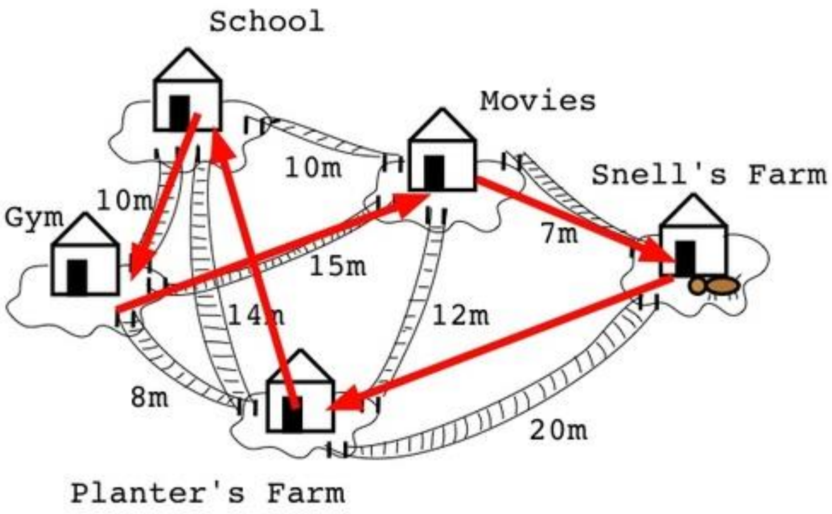
  3. Bila pencarian berakhir, umumkan solusi terbaik (*the winner*)

- Meskipun *exhaustive search* secara teoritis menghasilkan solusi, namun waktu atau sumberdaya yang dibutuhkan dalam pencarian solusinya sangat besar.

Contoh-contoh *exhaustive search*

1. Travelling Salesperson Problem (TSP)

Persoalan: Diberikan *n* buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) dengan jarak terpendek yang dilakukan oleh seorang pedagang sehingga ia melalui setiap kota tepat hanya sekali dan kembali lagi ke kota asal keberangkatan.

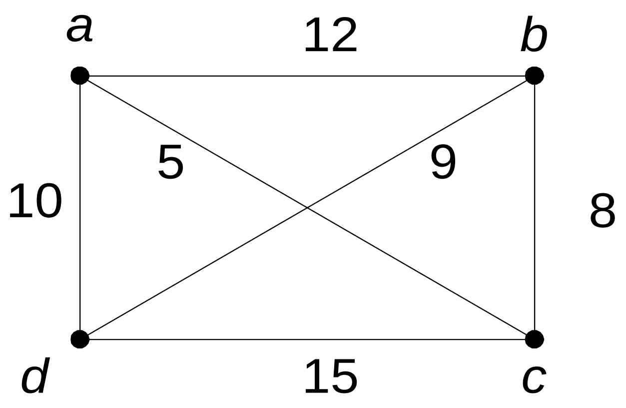


- Persoalan *TSP* tidak lain adalah menemukan sirkuit Hamilton dengan bobot minimum.

- 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. Itulah solusinya.

Contoh 3:

TSP dengan n = 4, simpul awal = a



No.	Rute perjalanan (*tour*)	Bobot
1.	$a \to b \to c \to d \to a$	12+8+15+10 = 45
2.	$a \to b \to d \to c \to a$	12+9+15+5 = 41
3.	$\mathbf{a \to c \to b \to d \to a}$	5+8+9+10 = **32** → optimal
4.	$a \to c \to d \to b \to a$	5+15+9+12 = 41
5.	$\mathbf{a \to d \to b \to c \to a}$	10+9+8+5 = **32** → optimal
6	$a \to d \to c \to b \to a$	10+15+8+12 = 45

Rute perjalananan terpendek adalah

$a \to c \to b \to d \to a$

$a \to d \to b \to c \to a$

dengan bobot = 32.

- Untuk graf lengkap n buah simpul semua rute perjalanan dibangkitkan dengan permutasi dari n – 1 buah simpul.

- Permutasi n – 1 buah simpul adalah

$$(n - 1)!$$

No.	Rute perjalanan (*tour*)	Bobot
1.	$a \to b \to c \to d \to a$	12+8+15+10 = 45
2.	$a \to b \to d \to c \to a$	12+9+15+5 = 41
3.	$\boldsymbol{a \to c \to b \to d \to a}$	5+8+9+10 = **32**
4.	$a \to c \to d \to b \to a$	5+15+9+12 = 41
5.	$\boldsymbol{a \to d \to b \to c \to a}$	10+9+8+5 = **32**
6	$a \to d \to c \to b \to a$	10+15+8+12 = 45

- Pada contoh di atas, untuk n = 4 akan terdapat

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

buah rute perjalanan.

- Jika TSP diselesaikan dengan *exhaustive search*, maka kita harus mengenumerasi sebanyak $(n - 1)!$ buah sirkuit Hamilton, menghitung bobot setiap sirkuitnya, lalu 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 semua rute perjalanan adalah hasil pencerminan dari setengah rute yang lain, yakni dengan mengubah arah rute perjalanan

1 dan 6

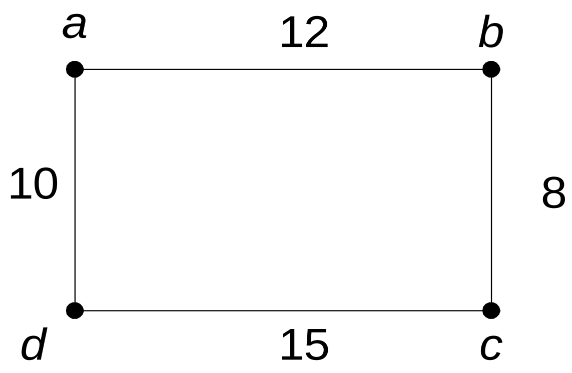
2 dan 4

3 dan 5

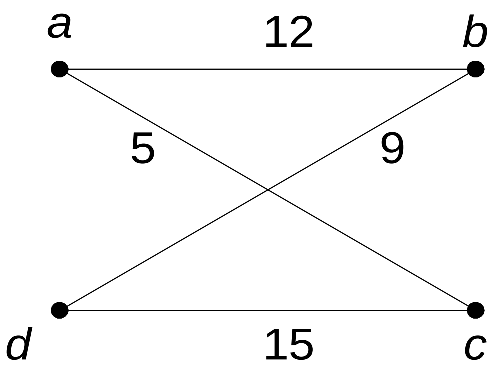
No.	Rute perjalanan (*tour*)	Bobot
1.	$a \to b \to c \to d \to a$	12+8+15+10 = 45
2.	$a \to b \to d \to c \to a$	12+9+15+5 = 41
3.	$\boldsymbol{a \to c \to b \to d \to a}$	5+8+9+10 = **32**
4.	$a \to c \to d \to b \to a$	5+15+9+12 = 41
5.	$\boldsymbol{a \to d \to b \to c \to a}$	10+9+8+5 = **32**
6	$a \to d \to c \to b \to a$	10+15+8+12 = 45

- maka dapat dihilangkan setengah dari jumlah permutasi (dari 6 menjadi 3).

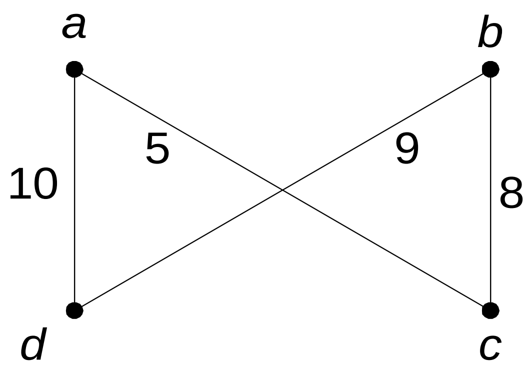
- Ketiga buah sirkuit Hamilton yang dihasilkan:



Bobot = 45



Bobot = 41



Bobot = 32

- Untuk graf lengkap dengan *n* buah simpul, kita hanya perlu mengevaluasi $(n – 1)!/2$ sirkuit Hamilton.

- Untuk TSP dengan n 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.

- Jika untuk mengevaluasi satu sirkuit Hamilton dibutuhkan waktu 1 detik, maka waktu yang dibutuhkan untuk mengevaluasi 6×10^{16} sirkuit Hamilton adalah sekitar 190 juta tahun

- Sayangnya, untuk persoalan TSP belum 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.

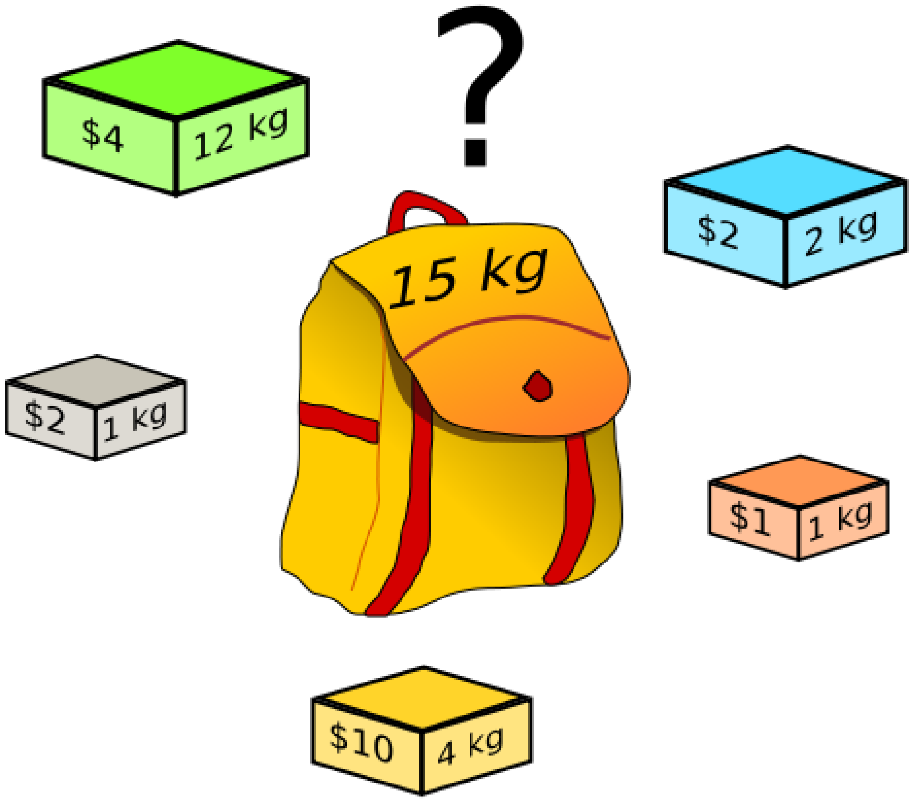
2. 1/0 Knapsack Problem

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

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

- Disebut *1/0 knapsack* problem karena suatu objek dapat dimasukkan ke dalam *knapsack* (1) atau tidak dimasukkan sama sekali (0)

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



- Solusi persoalan dinyatakan sebagai $X = \{x_1, x_2, \ldots, x_n\}$

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

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

- Formulasi persoalan *knapsack* secara matematis:

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

dengan kendala (*constraint*)

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

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

Algoritma *exhaustive search* untuk persoalan *1/0 Knapsack*:

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 namun total bobotnya tidak melebihi kapasitas *knapsack*.

Contoh 4: Misalkan terdapat n = 4 buah objek dan sebuah *knapsack* dengan kapasitas K = 16. Properti setiap objek adalah sbb

Objek	Bobot	Profit ($)
1	2	20
2	5	30
3	10	50
4	5	10

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** $\rightarrow$ **optimal**
{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: $X = \{0, 1, 1, 0\}$

- Banyaknya himpunan bagian dari sebuah himpunan dengan n elemen adalah 2^n.

  Waktu untuk menghitung total bobot di dalam himpunan bagian adalah $O(n)$

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

- TSP dan 0/1 *Knapsack*, adalah contoh persoalan dengan kompleksitas eksponensial.

Bersambung ke bagian 2