Algoritma *Brute Force* (lanjutan)

Contoh-contoh lain

1. Pencocokan String *(String Matching)*

Persoalan: Diberikan
a. teks (*text*), yaitu (*long*) *string* yang panjangnya *n* karakter
b. *pattern*, yaitu *string* dengan panjang *m* karakter ($m < n$) yang akan dicari di dalam teks.

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 NOTICED HIM
1 NOT
2  NOT
3   NOT
4    NOT
5     NOT
6      NOT
7       NOT
8        NOT
```

Contoh 2:
Pattern: 001011
Teks: 10010101**001011**110101010001

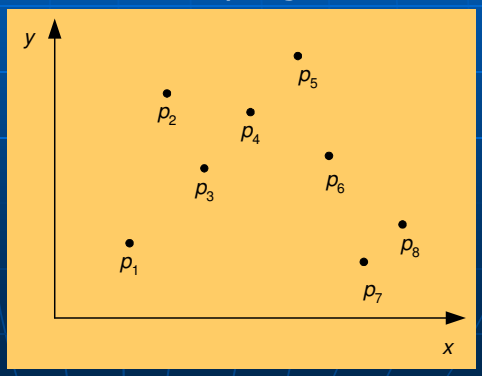
```
  10010101001011110101010001
1 001011
2  001011
3   001011
4    001011
5     001011
6      001011
7       001011
8        001011
9         001011
```

```
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 direpresentasika 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

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



- Jarak dua buah titik di bidang 2-D, $p_1 = (x_1, y_1)$ dan $p_2 = (x_2, y_2)$ adalah (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←√((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: $O(n^2)$.

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 sekontruktif/sekreatif teknik pemecahan masalah lainnya.

- Ken Thompson (salah seorang penemu Unix) mengatakan: "*When in doubt, use brute force*", faktanya kernel Unix yang asli lebih menyukai algoritma yang sederhana dan kuat (*robust*) daripada algoritma yang cerdas tapi rapuh.

Exhaustive Search

Exhaustive search adalah

- teknik pencarian solusi secara solusi *brute force* untuk masalah yang melibatkan pencarian elemen dengan sifat khusus;
- 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, mungkin saja beberapa kemungkinan solusi yang tidak layak dikeluarkan, dan 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 (TSP)*

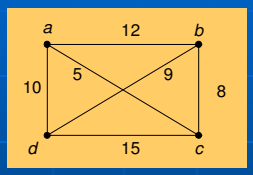
- Persoalan: Diberikan *n* buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) terpendek yang melalui setiap kota lainnya hanya sekali dan kembali lagi ke kota asal keberangkatan.
- Persoalan *TSP* tidak lain adalah menemukan sirkuit Hamilton dengan bobot minimum.

- Algoritma *exhaustive search* untuk persoalan 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 (*tour*)	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.	$\boldsymbol{a \rightarrow c \rightarrow b \rightarrow d \rightarrow a}$	10+5+9+8 = **32**
4.	$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	12+5+9+15 = 41
5.	$\boldsymbol{a \rightarrow d \rightarrow b \rightarrow c \rightarrow a}$	10+5+9+8 = **32**
6	$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	10+12+8+15 = 45

Rute perjalananan terpendek adalah
a→*c*→*b*→*d*→*a*
a→*d*→*b*→*c*→*a*
dengan bobot = 32.

- Untuk n buah simpul semua rute perjalanan yang mungkin 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

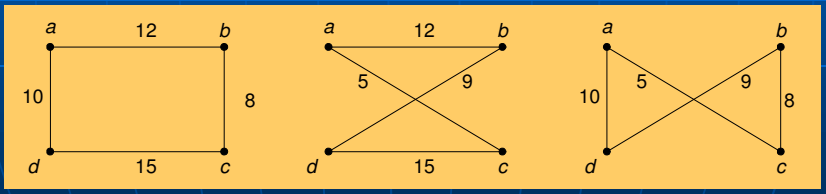
  $(4 - 1)! = 3! = 6$

  buah rute perjalanan.

- Jika diselesaikan dengan metode *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 adalah seperti gambar di bawah ini:



- Dengan demikian, untuk graf dengan *n* buah simpul, kita hanya perlu mengevaluasi sirkuit Hamilton sebanyak $(n - 1)!/2$ buah.
- Untuk ukuran masukan yang besar, algoritma *exhaustive search* menjadi sangat tidak mangkus.
- Pada persoalan *TSP* misalnya, untuk jumlah simpul $n = 20$ akan terdapat $(19!)/2 = 6 \times 1016$ 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 (*weigth*) w_i dan keuntungan(*profit*) p_i.

  Bagaimana memilih memilih objek-objek yang dimasukkan ke dalam *knapsack* sedemikian sehingga memaksimumkan 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 vektor *n*-tupel:

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

  x_i = 1 jika objek ke-*i* dimasukkan ke dalam *knapsack*,

  x_i = 0 jika objek ke-*i* tidak dimasukkan.

Formulasi 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 0/1 *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.

- **Contoh**: $n = 4$.

  $w_1 = 2$; $p_1 = 20$
  $w_2 = 5$; $p_2 = 30$
  $w_3 = 10$; $p_3 = 50$
  $w_4 = 5$; $p_4 = 10$
  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* = $O(n.\ 2^n)$.

- 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.

Exhaustive Search dalam Bidang Kriptografi

- Di dalam bidang 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

  $(2)(2)(2)(2)(2) \ldots (2)(2) = 2^{56} =$ 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!

- Meskipun algoritma *exhaustive search* tidak mangkus, namun – sebagaimana ciri algoritma *brute force* pada umumnya– 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 adalah teknik **heuristik** (*heuristic*).
- Teknik heuristik digunakan untuk mengeliminasi beberapa kemungkinan solusi tanpa harus mengeksplorasinya secara penuh. Selain itu, teknik heuristik juga membantu memutuskan kemungkinan solusi mana yang pertama kali perlu dievaluasi.

- 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 karena heuristik berlaku sebagai panduan (*guideline*), sedangkan algoritma adalah urutan langkah-langkah penyelesaian.
- Heuristik mungkin tidak selalu memberikan hasil yang diinginkan, tetapi secara ekstrim ia bernilai 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 masalah, tetapi seringkali memecahkan masalah dengan cukup baik untuk kebanyakan masalah, dan seringkali pula lebih cepat daripada pencarian solusi secara lengkap.
- Sudah sejak lama heuristik digunakan secara intensif di dalam bidang intelijensia 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 memerika 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.