Algoritma *Greedy*

Bahan Kuliah

IF2211 Strategi Algoritma

Oleh: Rinaldi Munir

Program Studi Informatika
Sekolah Teknik Elektro dan Informatika ITB

Pendahuluan

- Algoritma *greedy* merupakan metode yang paling populer untuk memecahkan persoalan optimasi.

- Persoalan optimasi (*optimization problems*):
  ➔ persoalan mencari solusi optimum.

- Hanya ada dua macam persoalan optimasi:
  1. Maksimasi (*maximization*)
  2. Minimasi (*minimization*)

Contoh persoalan optimasi:

(Persoalan Penukaran Uang):
Diberikan uang senilai *A*. Tukar *A* dengan koin-koin uang yang ada. Berapa jumlah minimum koin yang diperlukan untuk penukaran tersebut?

➔ Persoalan minimasi

Contoh 1: tersedia banyak koin 1, 5, 10, 25

- Uang senilai *A* = 32 dapat ditukar dengan banyak cara berikut:

  32 = 1 + 1 + … + 1 (32 koin)

  32 = 5 + 5 + 5 + 5 + 10 + 1 + 1 (7 koin)

  32 = 10 + 10 + 10 + 1 + 1 (5 koin)

  … dst

- Minimum: 32 = 25 + 5 + 1 + 1 (4 koin)

- *Greedy* = rakus, tamak, loba, …
- Prinsip *greedy*: "*take what you can get now!*".
- Algoritma *greedy* membentuk solusi langkah per langkah (*step by step*).
- Pada setiap langkah, terdapat banyak pilihan yang perlu dievaluasi.
- Oleh karena itu, pada setiap langkah harus dibuat keputusan yang terbaik dalam menentukan pilihan.

- Pada setiap langkah, kita membuat pilihan **optimum lokal** (*local optimum*)

- dengan harapan bahwa langkah sisanya mengarah ke solusi **optimum global** (*global optimm*).

- Algoritma *greedy* adalah algoritma yang memecahkan masalah langkah per langkah;

pada setiap langkah:

1. mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan (prinsip "*take what you can get now!*")

2. berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global.

- Tinjau masalah penukaran uang:

  Strategi *greedy*:

  Pada setiap langkah, pilihlah koin dengan nilai terbesar dari himpunan koin yang tersisa.

- Misal: A = 32, koin yang tersedia: 1, 5, 10, dan 25
  Langkah 1: pilih 1 buah koin 25 (Total = 25)
  Langkah 2: pilih 1 buah koin 5 (Total = 25 + 5 = 30)
  Langkah 3: pilih 2 buah koin 1 (Total = 25+5+1+1= 32)

- Solusi: Jumlah koin minimum = 4 (solusi optimal!)

Elemen-elemen algoritma greedy:

1. Himpunan kandidat, *C*.
2. Himpunan solusi, *S*
3. Fungsi seleksi (*selection function*)
4. Fungsi kelayakan (*feasible*)
5. Fungsi obyektif

Dengan kata lain:

algoritma *greedy* melibatkan pencarian sebuah himpunan bagian, *S*, dari himpunan kandidat, *C*;

yang dalam hal ini, *S* harus memenuhi beberapa kriteria yang ditentukan, yaitu menyatakan suatu solusi dan *S* dioptimisasi oleh fungsi obyektif.

Pada masalah penukaran uang:

- *Himpunan kandidat*: himpunan koin yang merepresentasikan nilai 1, 5, 10, 25, paling sedikit mengandung satu koin untuk setiap nilai.
- *Himpunan solusi*: total nilai koin yang dipilih tepat sama jumlahnya dengan nilai uang yang ditukarkan.
- *Fungsi seleksi*: pilihlah koin yang bernilai tertinggi dari himpunan kandidat yang tersisa.
- *Fungsi layak*: memeriksa apakah nilai total dari himpunan koin yang dipilih tidak melebihi jumlah uang yang harus dibayar.
- *Fungsi obyektif*: jumlah koin yang digunakan minimum.

Skema umum algoritma *greedy*:

```
function greedy(input C: himpunan_kandidat)→ himpunan_kandidat
{ Mengembalikan solusi dari persoalan optimasi dengan algoritma greedy
  Masukan: himpunan kandidat C
  Keluaran: himpunan solusi yang bertipe himpunan_kandidat
}
Deklarasi
    x : kandidat
    S : himpunan_kandidat

Algoritma:
    S ← {}   { inisialisasi S dengan kosong }
    while (not SOLUSI(S)) and (C ≠ {} ) do
       x ← SELEKSI(C)          { pilih sebuah kandidat dari C}
       C ← C - {x}             { elemen himpunan kandidat berkurang satu }
       if LAYAK(S ∪ {x}) then
           S ← S ∪ {x}
       endif
    endwhile
    {SOLUSI(S) or C = {} }

    if SOLUSI(S) then
       return S
    else
       write('tidak ada solusi')
    endif
```

- Pada akhir setiap lelaran, solusi yang terbentuk adalah optimum lokal.
- Pada akhir kalang while-do diperoleh optimum global.

- *Warning*: Optimum global belum tentu merupakan solusi optimum (terbaik), tetapi *sub-optimum* atau *pseudo-optimum*.

- Alasan:
  1. Algoritma *greedy* tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada (sebagaimana pada metode *exhaustive search)*.

  2. Terdapat beberapa fungsi SELEKSI yang berbeda, sehingga kita harus memilih fungsi yang tepat jika kita ingin algoritma menghasilkan solusi optiamal.

- Jadi, pada sebagian masalah algoritma *greedy* tidak selalu berhasil memberikan solusi yang optimal.

- **Contoh 2**: tinjau masalah penukaran uang.

(a) Koin: 5, 4, 3, dan 1
Uang yang ditukar = 7.
Solusi *greedy*: 7 = 5 + 1 + 1 (3 koin) → tidak optimal
Solusi optimal: 7 = 4 + 3 (2 koin)

(b) Koin: 10, 7, 1
Uang yang ditukar: 15
Solusi *greedy*: 15 = 10 + 1 + 1 + 1 + 1 + 1 (6 koin)
Solusi optimal: 15 = 7 + 7 + 1 (hanya 3 koin)

(c) Koin: 15, 10, dan 1
Uang yang ditukar: 20
Solusi *greedy*: 20 = 15 + 1 + 1 + 1 + 1 + 1 (6 koin)
Solusi optimal: 20 = 10 + 10 (2 koin)

- Untuk sistem mata uang dollar AS, euro Eropa, dan *crown* Swedia, algoritma *greedy* selalu memberikan solusi optimum.

- Contoh: Uang $6,39 ditukar dengan uang kertas (*bill*) dan koin sen (*cent*), kita dapat memilih:
  - Satu buah uang kertas senilai $5
  - Satu buah uang kertas senilai $1
  - Satu koin 25 sen
  - Satu koin 10 sen
  - Empat koin 1 sen

$5 + $1 + 25c + 10c + 1c + 1c + 1c + 1c = $6,39

- Jika jawaban terbaik mutlak tidak diperlukan, maka algoritma *greedy* sering berguna untuk menghasilkan solusi hampiran (*approximation*),

- daripada menggunakan algoritma yang lebih rumit untuk menghasilkan solusi yang eksak.

- Bila algoritma *greedy* optimum, maka keoptimalannya itu dapat dibuktikan secara matematis

Contoh-contoh Algoritma Greedy

1. **Masalah penukaran uang**

   Nilai uang yang ditukar: A

   Himpunan koin (*multiset*): $\{d_1, d_2, \ldots\}$.

   Himpunan solusi: $X = \{x_1, x_2, \ldots, x_n\}$,

   $x_i = 1$ jika d_i dipilih, $x_i = 0$ jika d_i tidak dipilih.

Obyektif persoalan adalah

Minimisasi $F = \sum_{i=1}^{n} x_i$ (fungsi obyektif)

dengan kendala $\sum_{i=1}^{n} d_i x_i = A$

Penyelesaian dengan *exhaustive search*

- Terdapat 2^n kemungkinan solusi
  (nilai-nilai $X = \{x_1, x_2, \ldots, x_n\}$)
- Untuk mengevaluasi fungsi obyektif = $O(n)$
- Kompleksitas algoritma *exhaustive search* seluruhnya = $O(n \cdot 2^n)$.

Penyelesaian dengan algoritma *greedy*

- Strategi *greedy*: Pada setiap langkah, pilih koin dengan nilai terbesar dari himpunan koin yang tersisa.

```
function CoinExchange(input C : himpunan_koin, A : integer) → himpunan_koin
{ mengembalikan koin-koin yang total nilainya = A, tetapi jumlah koinnya minimum }

Deklarasi
   S : himpunan_koin
   x : koin

Algoritma
  S ← {}
  while (∑(nilai semua koin di dalam S) ≠ A) and (C ≠ {} ) do
     x ← koin yang mempunyai nilai terbesar
     C ← C - {x}
     if  (∑(nilai semua koin di dalam S) + nilai koin x ≤ A then
       S ← S ∪ {x}
     endif
  endwhile

  if (∑(nilai semua koin di dalam S) = A then
     return S
  else
     write('tidak ada solusi')
  endif
```

- Agar pemilihan koin berikutnya optimal, maka perlu mengurutkan himpunan koin dalam urutan yang menurun (*noninceasing order*).

- Jika himpunan koin sudah terurut menurun, maka kompleksitas algoritma *greedy* = $O(n)$.

- Sayangnya, algoritma *greedy* untuk masalah penukaran uang ini tidak selalu menghasilkan solusi optimal (lihat contoh sebelumnya).

2. Minimisasi Waktu di dalam Sistem (Penjadwalan)

- **Persoalan**: Sebuah *server* (dapat berupa *processor*, pompa, kasir di bank, dll) mempunai *n* pelanggan (*customer*, *client*) yang harus dilayani. Waktu pelayanan untuk setiap pelanggan *i* adalah t_i.

  Minimumkan total waktu di dalam sistem:

  $T = \sum_{i=1}^{n}$ (waktu di dalam sistem)

- Ekivalen dengan meminimumkan waktu rata-rata pelanggan di dalam sistem.

Contoh 3: Tiga pelanggan dengan

$t_1 = 5$, $t_2 = 10$, $t_3 = 3$,

Enam urutan pelayanan yang mungkin:

Urutan	*T*
1, 2, 3:	5 + (5 + 10) + (5 + 10 + 3) = 38
1, 3, 2:	5 + (5 + 3) + (5 + 3 + 10) = 31
2, 1, 3:	10 + (10 + 5) + (10 + 5 + 3) = 43
2, 3, 1:	10 + (10 + 3) + (10 + 3 + 5) = 41
3, 1, 2:	**3 + (3 + 5) + (3 + 5 + 10) = 29 ← (optimal)**
3, 2, 1:	3 + (3 + 10) + (3 + 10 + 5) = 34

Penyelesaian dengan *Exhaustive Search*

- Urutan pelangan yang dilayani oleh *server* merupakan suatu permutasi
- Jika ada *n* orang pelanggan, maka tedapat *n*! urutan pelanggan
- Untuk mengevaluasi fungsi obyektif : *O*(*n*)
- Kompleksitas algoritma *exhaustive search* = O(*nn*!)

Penyelesaian dengan algoritma *greedy*

- Strategi *greedy*: Pada setiap langkah, pilih pelanggan yang membutuhkan waktu pelayanan terkecil di antara pelanggan lain yang belum dilayani.

```
function PenjadwalanPelanggan(input C : himpunan_pelanggan) → himpunan_pelanggan
{ mengembalikan urutan jadwal pelayanan pelanggan yang meminimumkan waktu di dalam
sistem }

Deklarasi
   S : himpunan_pelanggan
   i : pelanggann

Algoritma
  S ← {}
  while (C ≠ {}) do
     i ← pelanggan yang mempunyai t[i] terkecil
     C ← C - {i}
     S ← S ∪ {i}
  endwhile

  return S
```

- Agar proses pemilihan pelanggan berikutnya optimal, urutkan pelanggan berdasarkan waktu pelayanan dalam urutan yang menaik.

- Jika pelanggan sudah terurut, kompleksitas algoritma *greedy* = $O(n)$.

```
procedure PenjadwalanPelanggan(input n:integer)

{ Mencetak informasi deretan pelanggan yang akan diproses oleh
server tunggal
  Masukan: n pelangan, setiap pelanggan dinomori 1, 2, …, n
  Keluaran: urutan pelanggan yang dilayani
}
Deklarasi
   i : integer

Algoritma:
  {pelanggan 1, 2, ..., n sudah diurut menaik berdasarkan t_i}
  for i←1 to n do
     write('Pelanggan ', i, ' dilayani!')
  endfor
```

- Algoritma *greedy* untuk penjadwalan pelanggan akan selalu menghasilkan solusi optimum.

- **Teorema.** Jika $t_1 \leq t_2 \leq \ldots \leq t_n$ maka pengurutan $i_j = j,\ 1 \leq j \leq n$ meminimumkan

$$T = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j}$$

untuk semua kemungkinan permutasi i_j.

3. *An Activity Selection Problem*

- **Persoalan**: Misalkan kita memiliki $S = \{1, 2, \ldots, n\}$ yang menyatakan n buah aktivitas yang ingin menggunakan sebuah *resource*, misalnya ruang pertemuan, yang hanya dapat digunakan satu aktivitas setiap saat.

  Tiap aktivitas i memiliki waktu mulai s_i dan waktu selesai f_i, dimana $s_i \leq f_i$. Dua aktivitas i dan j dikatakan **kompatibel** jika interval $[s_i, f_i]$ dan $[s_j, f_j]$ tidak bentrok.

  Masalah *Activity selection problem* ialah memilih sebanyak mungkin aktivitas yang bisa dilayani.

Contoh Instansiasi Persoalan:

i	s_i	f_i
1	1	4
2	3	5
3	4	6
4	5	7
5	3	8
6	7	9
7	10	11
8	8	12
9	8	13
10	2	14
11	13	15

Penyelesaian dengan *Exhaustive Search*

- Tentukan semua himpunan bagian dari himpunan dengan *n* aktivitas.
- Evaluasi setiap himpunan bagian apakah semua aktivitas di dalamnya kompatibel.
- Jika kompatibel, maka himpunan bagian tersebut adalah solusinya
- Kompleksitas waktu algoritmanya $O(2^n)$

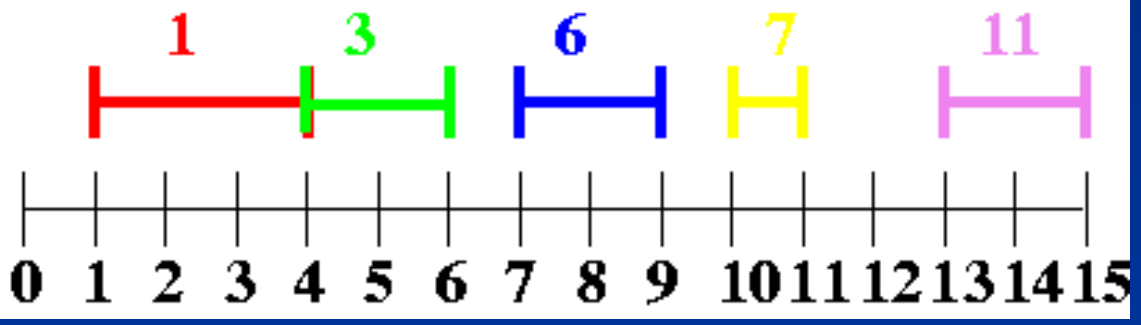
- Apa strategi *greedy-nya*?

  1. Urutkan semua aktivitas berdasarkan waktu selesai dari kecil ke besar

  2. Pada setiap step, pilih aktivitas yang waktu mulainya lebih besar atau sama dengan waktu selesai aktivitas yang dipilih sebelumnya

i	si	fi
1	1	4
2	3	5
3	4	6
4	5	7
5	3	8
6	7	9
7	10	11
8	8	12
9	8	13
10	2	14
11	13	15

$A = \{1\}, fj = 4$
$A = \{1,3\}, fj = 6$
$A = \{1,3,6\}, fj = 9$
$A = \{1,3,6,7\}, fj = 11$
$A = \{1,3,6,7,11\}, fj = 15$



Solusi: aktivitas yang dipilih adalah 1, 3, 6, 7, dan 11

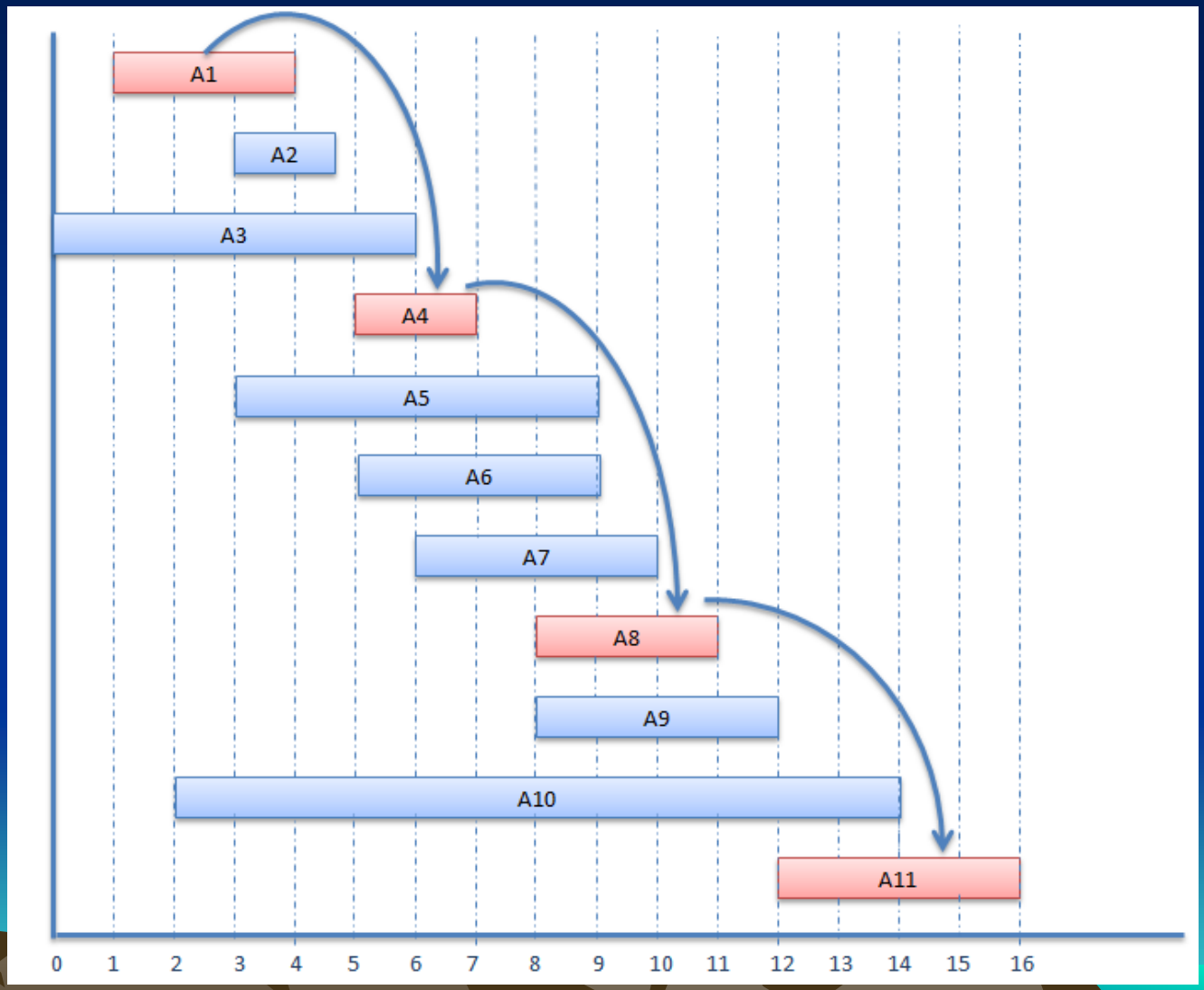
- Contoh lain:

Example

Start time (si)	finish time (fi)	Activity name
1	4	A1
3	5	A2
0	6	A3
5	7	A4
3	9	A5
5	9	A6
6	10	A7
8	11	A8
8	12	A9
2	14	A10
12	16	A11

Sumber: http://scanftree.com/Data_Structure/activity-selection-problem

- Solusi dengan Algoritma *Greedy*:



Algoritma

```
function Greedy-Activity-Selector(s, f)
{ Asumsi: aktivitas sudah diurut terlebih dahulu
Berdasarkan waktu selesai: f1 <= f2 <= … <= fn }

Algoritma
   n ← length(s)
   A ← {1}
   j ← 1
   for i ← 2 to n do
       if si >= fj then
           A ← A Ụ {i]
           j ← i
       end
   end
```

Kompleksitas algoritma: O(n) {jika waktu pengurutan tidak diperhtungkan}

Bukti:

The greedy-choice property:

There exists an optimal solution A such that the greedy choice "1" in A.

The proof goes as follows:

- let's order the activities in A by finish time such that the first activity in A is "k_1".
- If $k_1 = 1$, then A begins with a greedy choice
- If $k_1 \neq 1$, then let $A' = (A - \{k_1\}) \cup \{1\}$.
  Then
  1. the sets $A - \{k_1\}$ and $\{1\}$ are disjoint
  2. the activities in A' are compatible
  3. A' is also optimal, since $|A'| = |A|$
- Therefore, we conclude that there always exists an optimal solution that begins with a greedy choice.

Sumber: http://web.cs.ucdavis.edu/~bai/ECS122A/ActivitySelect.pdf

- Usulan strategi *greedy* yang lain: pilih aktivitas yang durasinya paling kecil lebih dahulu dan waktu mulainya tidak lebih besar dari waktu selesai aktivitas lain yang telah terpilih

➔lebih rumit

i	s_i	f_i	durasi
1	1	4	3
2	3	5	**2**
3	4	6	2
4	5	7	**2**
5	3	8	5
6	7	9	**2**
7	10	11	**1**
8	8	12	4
9	8	13	5
10	2	14	12
11	13	15	**2**

Solusi: aktivitas 7, 2, 4, 6, dan 11

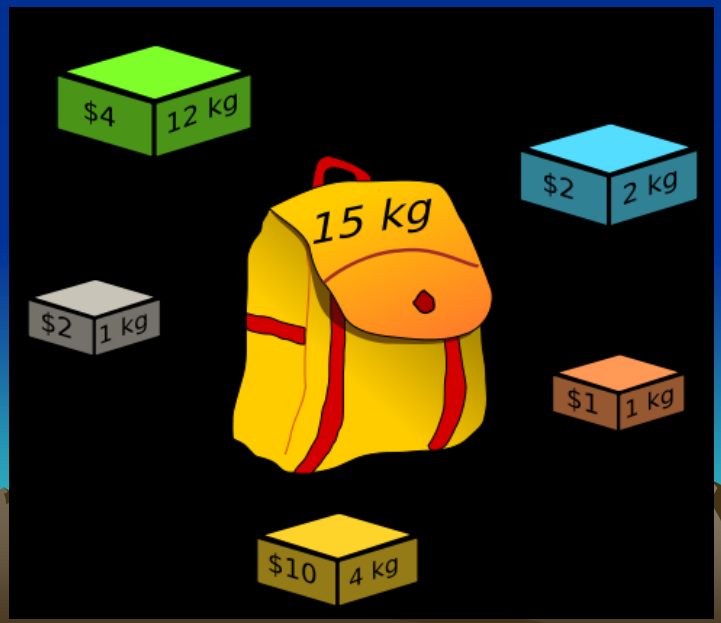
4. *Integer Knapsack*

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, $i = 1, 2, \ldots, n$



Penyelesaian dengan *exhaustive search*

- Sudah dijelaskan pada pembahasan exhaustive search.

- Kompleksitas algoritma *exhaustive search* untuk persoalan ini = $O(n \cdot 2^n)$.

Penyelesaian dengan algoritma *greedy*

- Masukkan objek satu per satu ke dalam *knapsack*. Sekali objek dimasukkan ke dalam *knapsack*, objek tersebut tidak bisa dikeluarkan lagi.

- Terdapat beberapa strategi *greedy* yang heuristik yang dapat digunakan untuk memilih objek yang akan dimasukkan ke dalam *knapsack*:

1. *Greedy by profit*.

   - Pada setiap langkah, pilih objek yang mempunyai keuntungan terbesar.
   - Mencoba memaksimumkan keuntungan dengan memilih objek yang paling menguntungkan terlebih dahulu.

2. *Greedy by weight.*

   - Pada setiap langkah, pilih objek yang mempunyai berat teringan.
   - Mencoba memaksimumkan keuntungan dengan dengan memasukkan sebanyak mungkin objek ke dalam *knapsack*.

3. *Greedy by density*.

- Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai p_i / w_i terbesar.
- Mencoba memaksimumkan keuntungan dengan memilih objek yang mempunyai keuntungan per unit berat terbesar.

- Pemilihan objek berdasarkan salah satu dari ketiga strategi di atas tidak menjamin akan memberikan solusi optimal.

Contoh 4.

*w*1 = 6; *p*1 = 12; *w*2 = 5; *p*1 = 15;

*w*3 = 10; *p*1 = 50; *w*4 = 5; *p*1 = 10

Kapasitas *knapsack K* = 16

Properti objek				*Greedy by*			Solusi Optimal
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*	
1	6	12	2	0	1	0	0
2	5	15	3	1	1	1	1
3	10	50	5	1	0	1	1
4	5	10	2	0	1	0	0
Total bobot				15	16	15	15
Total keuntungan				65	37	65	65

- Solusi optimal: *X* = (0, 1, 1, 0)
- *Greedy by profit* dan *greedy by density* memberikan solusi optimal!

Contoh 5.

*w*1 = 100; *p*1 = 40; *w*2 = 50; *p*2 = 35; *w*3 = 45; *p*3 = 18;
*w*4 = 20; *p*4 = 4; *w*5 = 10; *p*5 = 10; *w*6 = 5; *p*6 = 2
Kapasitas *knapsack K* = 100

Properti objek				*Greedy by*			Solusi Optimal
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*	
1	100	40	0,4	1	0	0	0
2	50	35	0,7	0	0	1	1
3	45	18	0,4	0	1	0	1
4	20	4	0,2	0	1	1	0
5	10	10	1,0	0	1	1	0
6	5	2	0,4	0	1	1	0
Total bobot				100	80	85	100
Total keuntungan				40	34	51	55

Ketiga strategi gagal memberikan solusi optimal!

Kesimpulan: Algoritma *greedy* tidak selalu berhasil menemukan solusi optimal untuk masalah 0/1 *Knapsack*.

4. Fractional Knapsack

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, $0 \leq x_i \leq 1$, $i = 1, 2, \ldots, n$

Penyelesaian dengan *exhaustive search*

- Oleh karena $0 \leq x_i \leq 1$, maka terdapat tidak berhinga nilai-nilai x_i.

- Persoalan *Fractional Knapsack* menjadi malar (*continuous*) sehingga tidak mungkin dipecahkan dengan algoritma *exhaustive search.*

Penyelesaian dengan algoritma *greedy*

- Ketiga strategi *greedy* yang telah disebutkan di atas dapat digunakan untuk memilih objek yang akan dimasukkan ke dalam *knapsack*.

- Mari kita bahas satu per satu

Contoh 6.

*w*1 = 18; *p*1 = 25; *w*2 = 15; *p*1 = 24

*w*3 = 10; *p*1 = 15 Kapasitas *knapsack K* = 20

Properti objek				*Greedy by*		
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*
1	18	25	1,4	1	0	0
2	15	24	1,6	2/15	2/3	1
3	10	15	1,5	0	1	1/2
Total bobot				20	20	20
Total keuntungan				28,2	31,0	31,5

- Solusi optimal: X = (0, 1, 1/2)
- yang memberikan keuntungan maksimum = 31,5.

- Strategi pemilihan objek ke dalam *knapsack* berdasarkan densitas p_i / w_i terbesar pasti memberikan solusi optimal.

- Agar proses pemilihan objek berikutnya optimal, maka kita urutkan objek berdasarkan p_i / w_i yang menurun, sehingga objek berikutnya yang dipilih adalah objek sesuai dalam urutan itu.

Teorema 3.2. Jika $p_1/w_1 \geq p_2/w_2 \geq ... \geq p_n/w_n$ maka algoritma *greedy* dengan strategi pemilihan objek berdasarkan p_i / w_i terbesar menghasilkan solusi yang optimum.

- Algoritma persoalan *fractional knapsack*:

1. Hitung harga p_i/w_i , i = 1, 2, ..., n
2. Urutkan seluruh objek berdasarkan nilai p_i/w_i dari besar ke kecil
3. Panggil `FractinonalKnapsack`

```
function FractionalKnapsack(input C : himpunan_objek, K : real) → himpunan_solusi

{ Menghasilkan solusi persoalan fractional knapsack dengan algoritma greedy yang
menggunakan strategi pemilihan objek berdasarkan density (pi/wi). Solusi dinyatakan
sebagai vektor X = x[1], x[2], …, x[n].

Asumsi: Seluruh objek sudah terurut berdasarkan nilai p_i/w_i yang menurun
}

Deklarasi
    i, TotalBobot : integer
    MasihMuatUtuh : boolean
    x : himpunan_solusi

Algoritma:
   for i ← 1 to n do
      x[i] ← 0   { inisialisasi setiap fraksi objek i dengan 0 }
   endfor

   i ← 0
   TotalBobot ← 0
   MasihMuatUtuh ← true
   while (i ≤ n) and (MasihMuatUtuh) do
     { tinjau objek ke-i }
     i ← i + 1
     if TotalBobot + C.w[i] ≤ K then
        { masukkan objek i ke dalam knapsack }
        x[i] ← 1
        TotalBobot ← TotalBobot + C.w[i]
     else
        MasihMuatUtuh ← false
        x[i] ← (K - TotalBobot)/C.w[i]
     endif
   endwhile
   { i > n or not MasihMuatUtuh }

   return x
```

Kompleksitas waktu algoritma = $O(n)$.

5. Penjadwalan *Job* dengan Tenggat Waktu (*Job Schedulling with Deadlines*)

Persoalan:

- Ada *n* buah *job* yang akan dikerjakan oleh sebuah mesin;
- tiap *job* diproses oleh mesin selama 1 satuan waktu dan tenggat waktu (*deadline*) setiap *job i* adalah $d_i \geq 0$;
- *job i* akan memberikan keuntungan sebesar p_i jika dan hanya jika *job* tersebut diselesaikan tidak melebihi tenggat waktunya;

- Bagaimana memilih *job-job* yang akan dikerjakan oleh mesin sehingga keuntungan yang diperoleh dari pengerjaan itu maksimum?

- Fungsi obyektif persoalan ini:

  Maksimasi $F = \sum_{i \in J} p_i$

- Solusi layak: himpunan *J* yang berisi urutan *job* yang sedemikian sehingga setiap *job* di dalam *J* selesai dikerjakan sebelum tenggat waktunya.

- Solusi optimum ialah solusi layak yang memaksimumkan *F*.

Contoh 7. Misalkan A berisi 4 *job* (n = 4):

$(p_1, p_2, p_3, p_4) = (50, 10, 15, 30)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Mesin mulai bekerja jam 6.00 pagi.

Job	Tenggat (d_i)	Harus selesai sebelum pukul
1	2 jam	8.00
2	1 jam	7.00
3	2 jam	8.00
4	1 jam	7.00

Pemecahan Masalah dengan *Exhaustive Search*

Cari himpunan bagian (*subset*) *job* yang layak dan memberikan total keuntungan terbesar.

Barisan *job*	Total keuntungan (*F*)	keterangan	
{}	0	layak	
{1}	50	layak	
{2}	10	layak	
{3}	15	layak	
{4}	30	layak	
{1, 2}	-	tidak layak	
{1, 3}	65	layak	
{1, 4}	-	tidak layak	
{2, 1}	60	layak	
{2, 3}	25	layak	
{2, 4}	-	tidak layak	
{3, 1}	65	layak	
{3, 2}	-	tidak layak	
{3, 4}	-	tidak layak	
{4, 1}	**80**	**layak**	**(Optimum!)**
{4, 2}	-	tidak layak	
{4, 3}	45	layak	

Pemecahan Masalah dengan Algoritma *Greedy*

- Strategi *greedy* untuk memilih *job*:

  Pada setiap langkah, pilih *job i* dengan p_i yang terbesar untuk menaikkan nilai fungsi obyektif *F*.

Contoh: $(p_1, p_2, p_3, p_4) = (50, 10, 15, 30)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

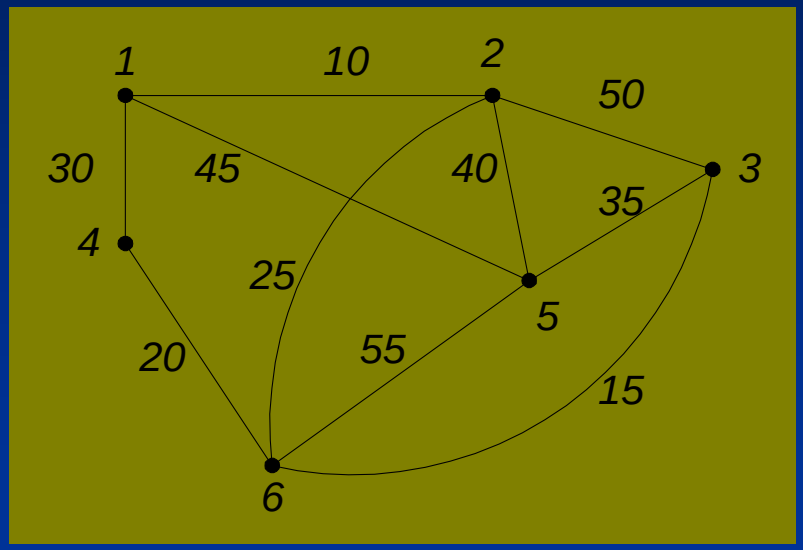
Langkah	J	$F = \sum p_i$	Keterangan
0	{}	0	-
1	{1}	50	layak
2	{4,1}	50 + 30 = 80	layak
3	{4, 1, 3}	-	tidak layak
4	{4, 1, 2}	-	tidak layak

Solusi optimal: J = {4, 1} dengan F = 80.

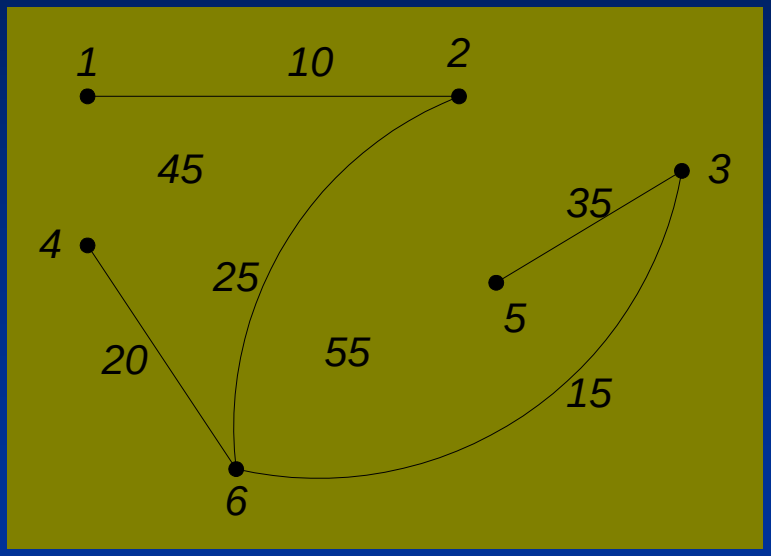
```
function JobSchedulling1(input C : himpunan_job) → himpunan_job
{ Menghasilkan barisan job yang akan diproses oleh mesin }

Deklarasi
  i : integer
  J : himpunan_job   { solusi }

Algoritma
  J ← {}
  while C ≠ {} do
     i ← job yang mempunyai p[i] terbesar
     C ← C - {i}
     if (semua job di dalam J ∪ {i} layak) then
        J ← J ∪ {i}
     endif
  endwhile
  { C = {} }
  return J
```

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

6. Pohon Merentang Minimum



(a) Graf $G = (V, E)$



(b) Pohon merentang minimum

(a) Algoritma Prim

- Strategi *greedy* yang digunakan:

  Pada setiap langkah, pilih sisi *e* dari graf *G*(*V*, *E*) yang mempunyai bobot terkecil dan bersisian dengan simpul-simpul di *T* tetapi *e* tidak membentuk sirkuit di *T*.

Algoritma Prim

Langkah 1: ambil sisi dari graf G yang berbobot minimum, masukkan ke dalam T.

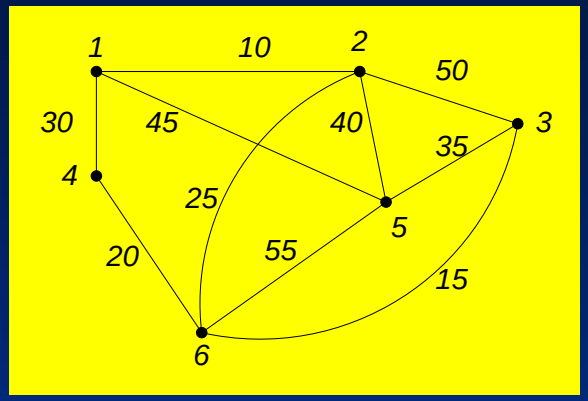
Langkah 2: pilih sisi (u, v) yang mempunyai bobot minimum dan bersisian dengan simpul di T, tetapi (u, v) tidak membentuk sirkuit di T. Masukkan (u, v) ke dalam T.

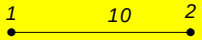
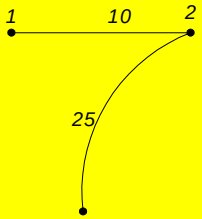
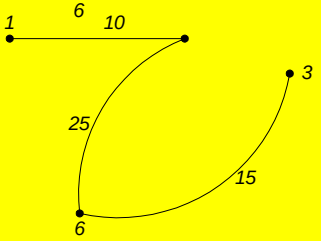
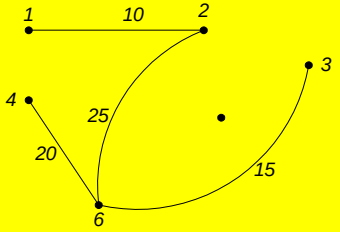
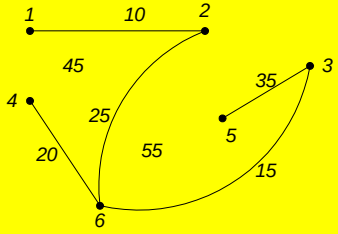
Langkah 3: ulangi langkah 2 sebanyak $n - 2$ kali.

```
procedure Prim(input G : graf, output T : pohon)
{ Membentuk pohon merentang minimum T dari graf terhubung-
berbobot G.
Masukan: graf-berbobot terhubung G = (V, E), dengan |V|= n
Keluaran: pohon rentang minimum T = (V, E')
}
Deklarasi
  i, p, q, u, v : integer

Algoritma
  Cari sisi (p,q) dari E yang berbobot terkecil
  T ← {(p,q)}
  for i←1 to n-2 do
    Pilih sisi (u,v) dari E yang bobotnya terkecil namun
    bersisian dengan simpul di T
    T ← T ∪ {(u,v)}
  endfor
```

Komplesitas algoritma: $O(n^2)$

Contoh:



Langkah	Sisi	Bobot	Pohon rentang
1	(1, 2)	10	
2	(2, 6)	25	
3	(3, 6)	15	
4	(4, 6)	20	
5	(3, 5)	35	

(a) Algoritma Kruskal

- Strategi *greedy* yang digunakan:

  Pada setiap langkah, pilih sisi *e* dari graf *G* yang mempunyai bobot minimum tetapi *e* tidak membentuk sirkuit di *T*.

Algoritma Kruskal

(Langkah 0: sisi-sisi dari graf sudah diurut menaik berdasarkan bobotnya – dari bobot kecil ke bobot besar)

Langkah 1: *T* masih kosong

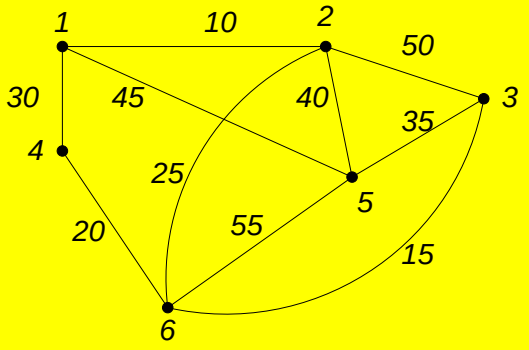
Langkah 2: pilih sisi (*u*, *v*) dengan bobot minimum yang tidak membentuk sirkuit di *T*. Tambahkan (*u*, *v*) ke dalam *T*.

Langkah 3: ulangi langkah 2 sebanyak $n - 1$ kali.

```
procedure Kruskal(input G : graf, output T : pohon)
{ Membentuk pohon merentang minimum T dari graf terhubung -
berbobot G.
Masukan: graf-berbobot terhubung G = (V, E), dengan |V|= n
Keluaran: pohon rentang minimum T = (V, E')
}
Deklarasi
  i, p, q, u, v : integer

Algoritma
  ( Asumsi: sisi-sisi dari graf sudah diurut menaik
    berdasarkan bobotnya - dari bobot kecil ke bobot
    besar)
  T ← {}
  while jumlah sisi T < n-1 do
    Pilih sisi (u,v) dari E yang bobotnya terkecil
    if (u,v) tidak membentuk siklus di T then
       T ← T ∪ {(u,v)}
    endif
  endfor
```

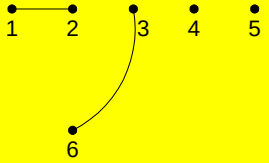
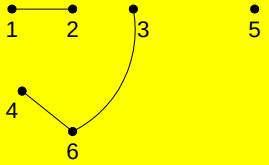
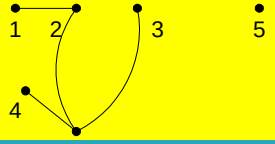
Kompleksitas algoritma: $O(|E| \log |E|)$

Contoh:

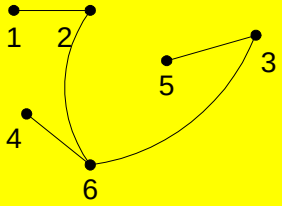


Sisi-sisi diurut menaik:

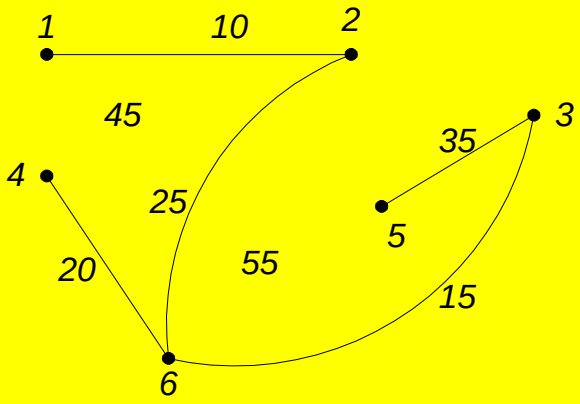
Sisi	(1,2)	(3,6)	(4,6)	(2,6)	(1,4)	(3,5)	(2,5)	(1,5)	(2,3)	(5,6)
Bobot	10	15	20	25	30	35	40	45	50	55

Langkah	Sisi	Bobot	Hutan merentang
0			
1	(1, 2)	10	
2	(3, 6)	15	
3	(4, 6)	20	
4	(2, 6)	25	

5	(1, 4)	30	ditolak
6	(3, 5)	35	



Pohon merentang minimum yang dihasilkan:



Bobot = 10 + 25 + 15 + 20 + 35 = 105

7. Lintasan Terpendek (*Shortest Path)*

Beberapa macam persoalan lintasan terpendek:

a) Lintasan terpendek antara dua buah simpul tertentu (*a pair shortest path*).

b) Lintasan terpendek antara semua pasangan simpul (*all pairs shortest path*).

c) Lintasan terpendek dari simpul tertentu ke semua simpul yang lain (*single-source shortest path*).

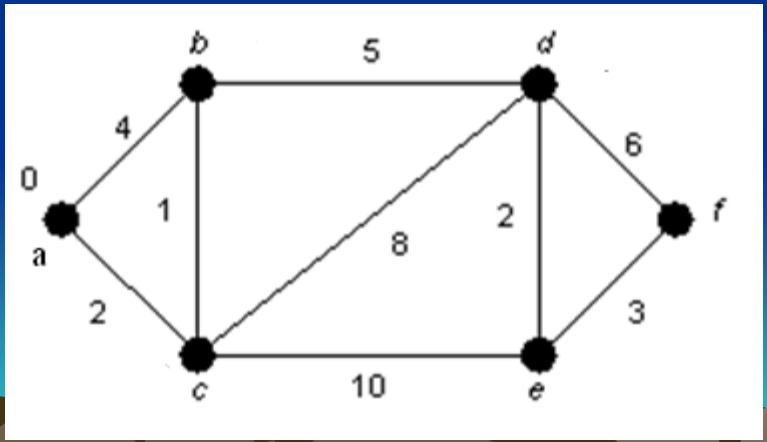
d) Lintasan terpendek antara dua buah simpul yang melalui beberapa simpul tertentu (*intermediate shortest path*).

→ Yang akan dibahas adalah persoalan c)

Persoalan:

Diberikan graf berbobot $G = (V, E)$. Tentukan lintasan terpendek dari sebuah simpul asal a ke setiap simpul lainnya di G.

Asumsi yang kita buat adalah bahwa semua sisi berbobot positif.



Berapa jarak terpendek berikut lintasannya dari:
- a ke b?
- a ke c?
- a ke d?
- a ke e?
- a ke f?

Penyelesaian dengan Algoritma *Brute Force*

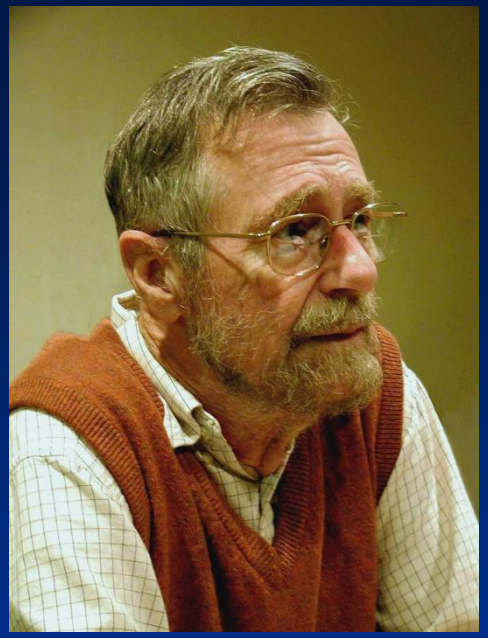
- Misalkan ingin menentukan jarak terpendek dari a ke b
- Enumerasi semua lintasan yang mungkin dibentuk dari a ke b, hitung panjangnya
- Lintasan yang memiliki panjang terkecil adalah lintasan terpendek dari a ke b
- Ulangi cara yang sama untuk jarak terpendek dari a ke c, dari a ke d, dan seterusnya.

Algoritma Dijkstra

Strategi *greedy*:

Pada setiap langkah, ambil sisi yang berbobot minimum yang menghubungkan sebuah simpul yang sudah terpilih dengan sebuah simpul lain yang belum terpilih.

Lintasan dari simpul asal ke simpul yang baru haruslah merupakan lintasan yang terpendek diantara semua lintasannya ke simpul-simpul yang belum terpilih.



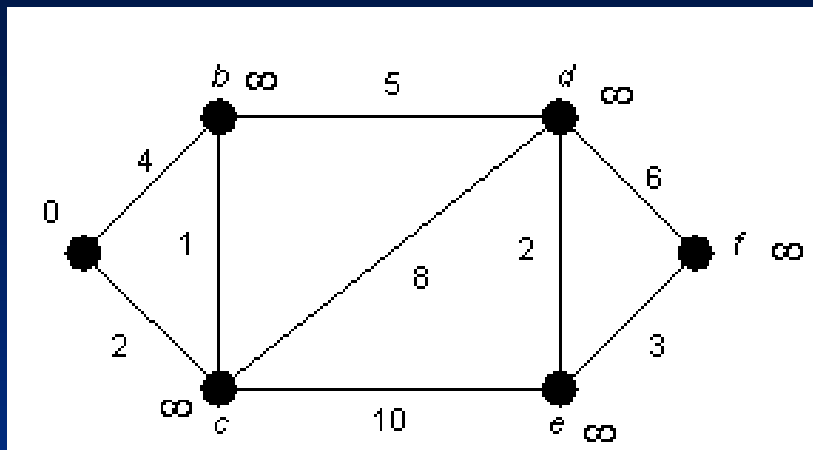
Edsger W. Dijkstra (1930–2002)

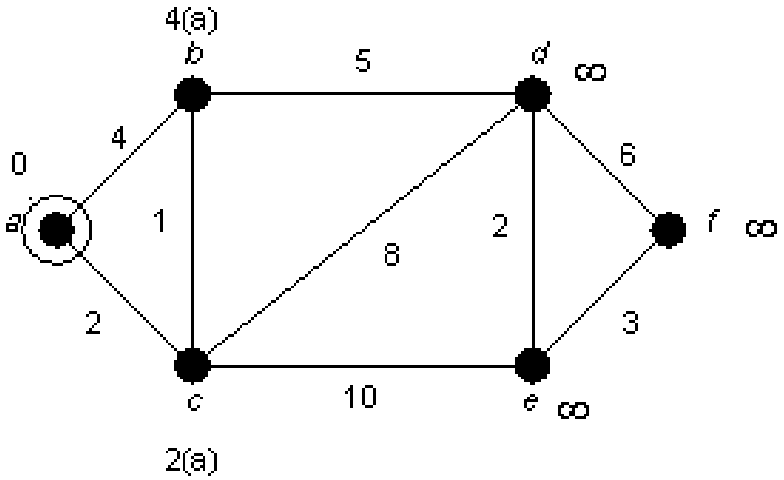
- Edsger Wybe Dijkstra was one of the most influential members of computing science's founding generation. Among the domains in which his scientific contributions are fundamental are
- algorithm design
- programming languages
- program design
- operating systems
- distributed processing

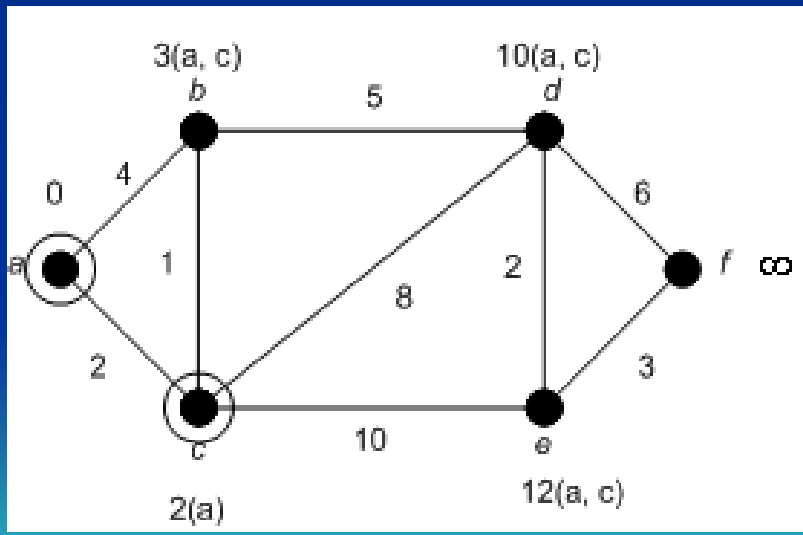
In addition, Dijkstra was intensely interested in teaching, and in the relationships between academic computing science and the software industry. During his forty-plus years as a computing scientist, which included positions in both academia and industry, Dijkstra's contributions brought him many prizes and awards, including computing science's highest honor, the ACM Turing Award.

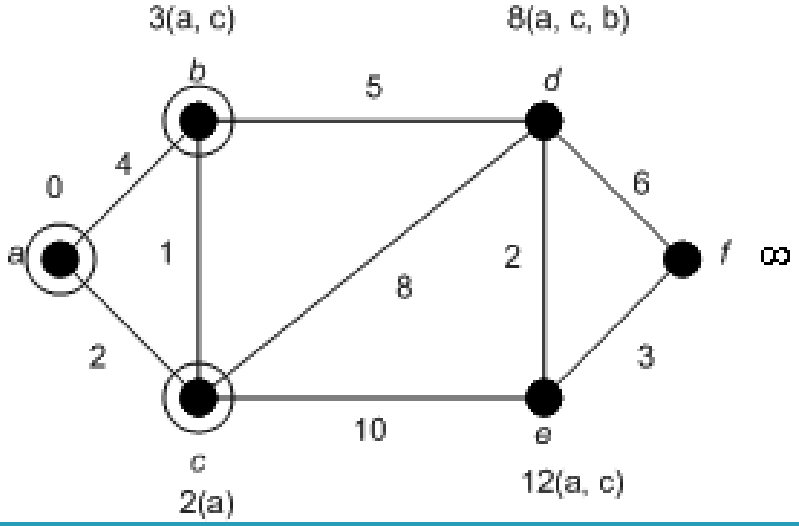
Sumber: http://www.cs.utexas.edu/users/EWD/

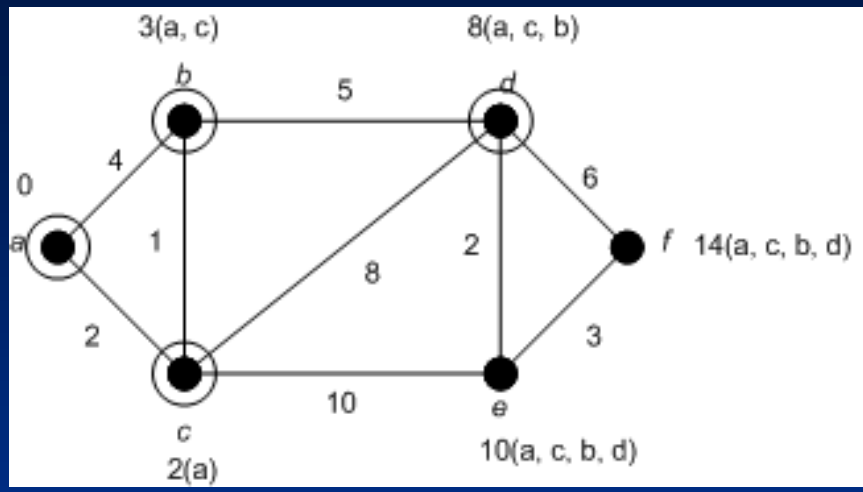
procedure Dijkstra (**input** G: weighted_graph, **input** a: intial_vertex)
Deklarasi:
    S : himpunan simpul solusi
    L : array[1..n] of real *{ L(z) berisi panjang lintasan terpendek dari a ke z}*
Algoritma
    for i←1 **to** n
        $L(v_i) \leftarrow \infty$
    end for
    L(a) ← 0; S ← { }
    while z ∉ S **do**
        u ← simpul yang bukan di dalam S dan memiliki L(u) minimum
        S ← S ∪ {u}
        for semua simpul v yang tidak terdapat di dalam S
            if L(u) + G(u,v) < L(v) then L(v) ← L(u) + G(u,v)
        end for
    end while

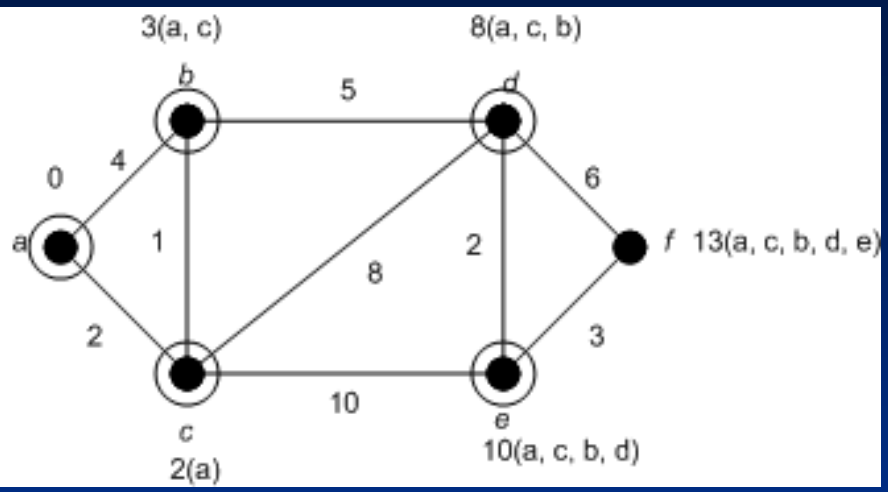

b ∞
5
d ∞
4
6
0
1
2
8
f ∞
2
3
∞
c
10
e ∞

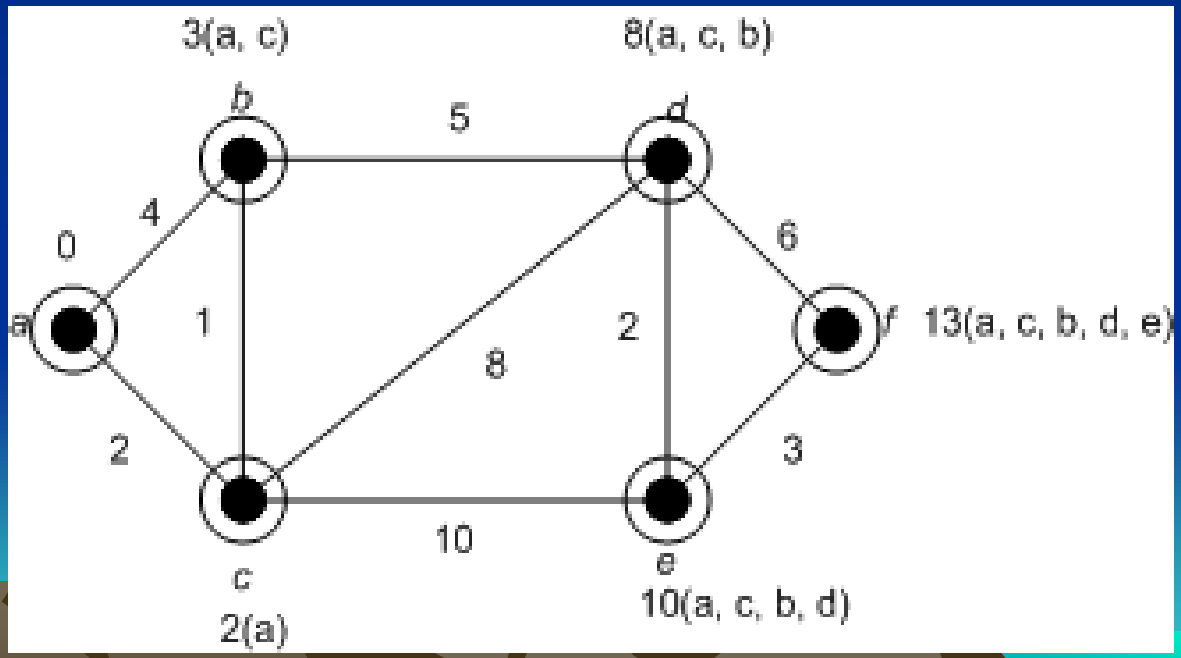

4(a)
b
5
d ∞
4
6
0
a
1
2
8
f ∞
2
3
c
10
e ∞
2(a)


3(a, c)
10(a, c)
b
5
d
0
4
6
a
1
2
8
f ∞
2
3
c
10
e
2(a)
12(a, c)


3(a, c)
8(a, c, b)
b
5
d
0
4
6
a
1
2
8
f ∞
2
3
c
10
e
2(a)
12(a, c)

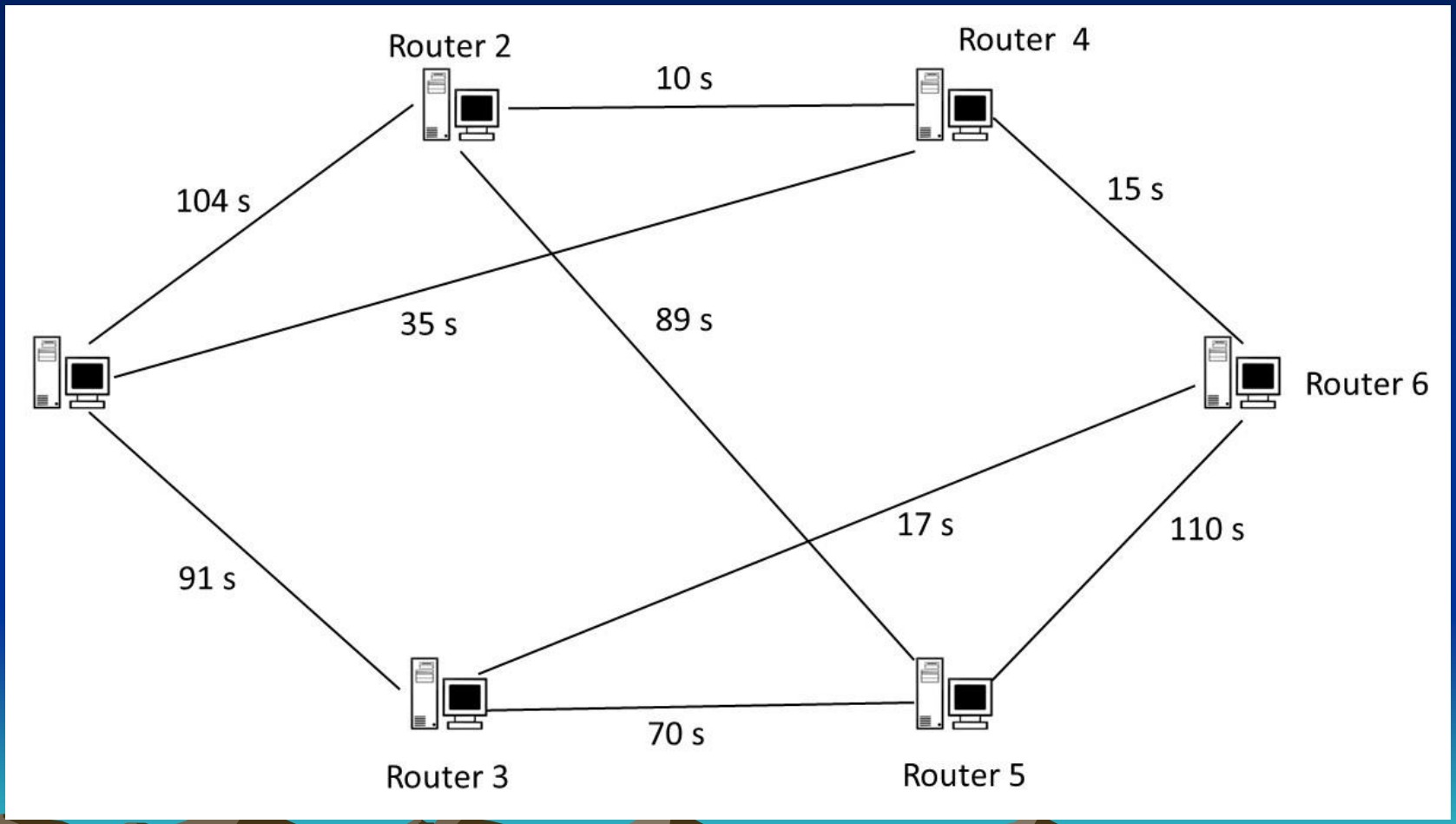

3(a, c)
8(a, c, b)
b
d
5
4
0
6
a
1
2
f
14(a, c, b, d)
8
2
3
10
e
c
2(a)
10(a, c, b, d)


3(a, c)
8(a, c, b)
b
d
5
4
0
6
a
1
2
f
13(a, c, b, d, e)
8
2
3
10
e
c
2(a)
10(a, c, b, d)


3(a, c)
8(a, c, b)
b
d
5
4
0
6
a
1
2
f
13(a, c, b, d, e)
8
2
3
10
e
c
2(a)
10(a, c, b, d)

Aplikasi algoritma Dijkstra:

→*Routing* pada jaringan komputer



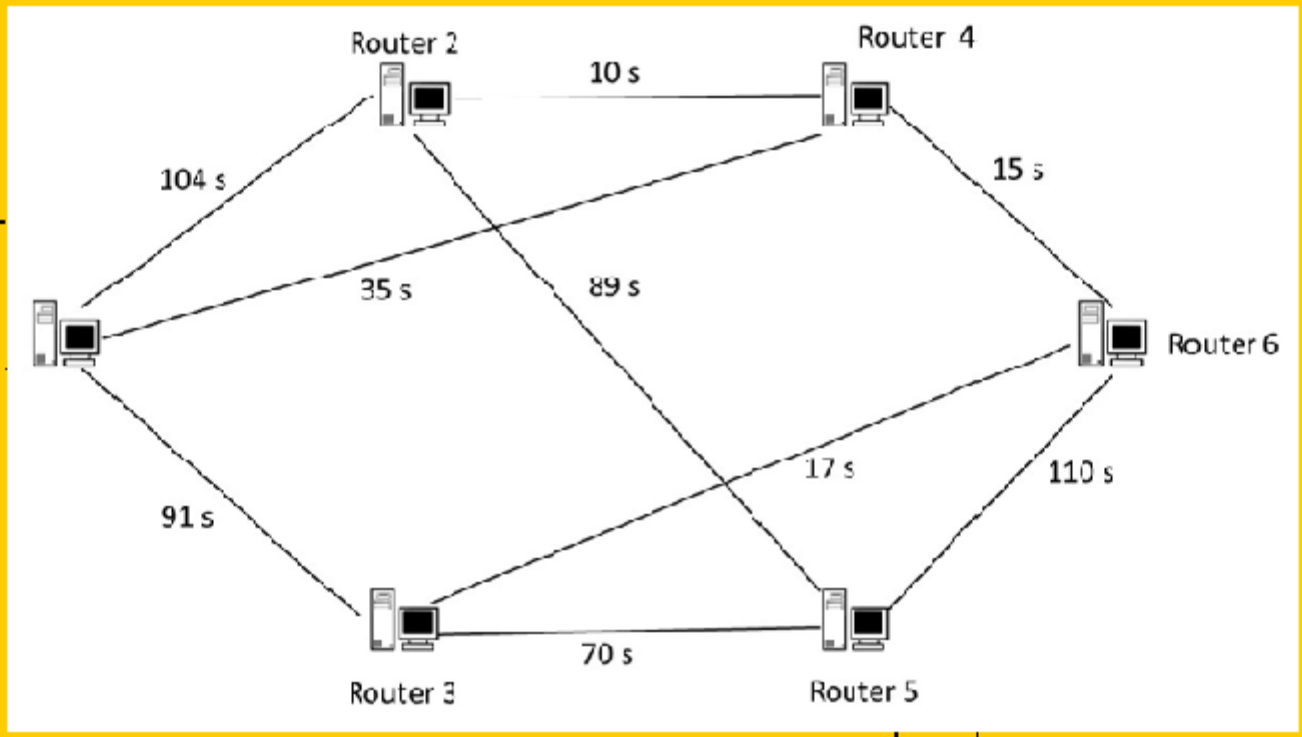
Lintasan terpendek (berdasarkan *delay time*):

Router Asal	*Router* Tujuan	Lintasan Terpendek
1	1	-
	2	1, 4, 2
	3	1, 4, 6, 3
	4	1, 4
	5	1, 4, 2, 5
	6	1, 4, 6
2	1	2, 4, 1
	2	-
	3	2, 4, 6, 3
	4	2, 4
	5	2, 5
	6	2, 4, 6
3	1	3, 6, 4, 1
	2	3, 6, 4, 2
	3	-
	4	3, 6, 4
	5	3, 5
	6	3, 6
4	1	4, 1
	2	4, 2
	3	4, 6, 2
	4	4, 6, 3
	5	4, 2, 5
	6	4, 6

Asal	Tujuan	Via
2	1	4
2	2	-
2	3	4
2	4	2
2	5	2
2	6	4

Asal	Tujuan	Via
4	1	4
4	2	4
4	3	6
4	4	-
4	5	2
4	6	4

Asal	Tujuan	Via
1	1	-
1	2	4
1	3	4
1	4	4
1	5	4
1	6	4



Asal	Tujuan	Via
6	1	4
6	2	4
6	3	6
6	4	6
6	5	3
6	6	-

Asal	Tujuan	Via
3	1	6
3	2	6
3	3	-
3	4	6
3	5	3
3	6	3

Asal	Tujuan	Via
5	1	2
5	2	5
5	3	5
5	4	2
5	5	-
5	6	3

8. **Pemampatan Data dengan Algoritma Huffman**

Prinsip kode Huffman:

- karakter yang paling sering muncul di dalam data dengan kode yang lebih pendek;
- sedangkan karakter yang relatif jarang muncul dikodekan dengan kode yang lebih panjang.

Fixed-length code

Karakter	*a*	*b*	*c*	*d*	*e*	*f*
Frekuensi	45%	13%	12%	16%	9%	5%
Kode	000	001	010	011	100	111

‘bad’ dikodekan sebagai ‘001000011’

Pengkodean 100.000 karakter membutuhkan 300.000 bit.

Variable-length code (Huffman code)

Karakter	*a*	*b*	*c*	*d*	*e*	*f*
Frekuensi	45%	13%	12%	16%	9%	5%
Kode	0	101	100	111	1101	1100

‘bad’ dikodekan sebagai ‘1010111 ’

Pengkodean 100.000 karakter membutuhkan

$(0{,}45 \times 1 + 0{,}13 \times 3 + 0{,}12 \times 3 + 0{,}16 \times 3 + 0{,}09 \times 4 + 0{,}05 \times 4) \times 100.000 = 224.000$ bit

Nisbah pemampatan:

$(300.000 – 224.000)/300.000 \times 100\% = 25{,}3\%$

Algoritma Greedy untuk Membentuk Kode Huffman:

1. Baca semua karakter di dalam data untuk menghitung frekuensi kemunculan setiap karakter. Setiap karakter penyusun data dinyatakan sebagai pohon bersimpul tunggal. Setiap simpul di-*assign* dengan frekuensi kemunculan karakter tersebut.

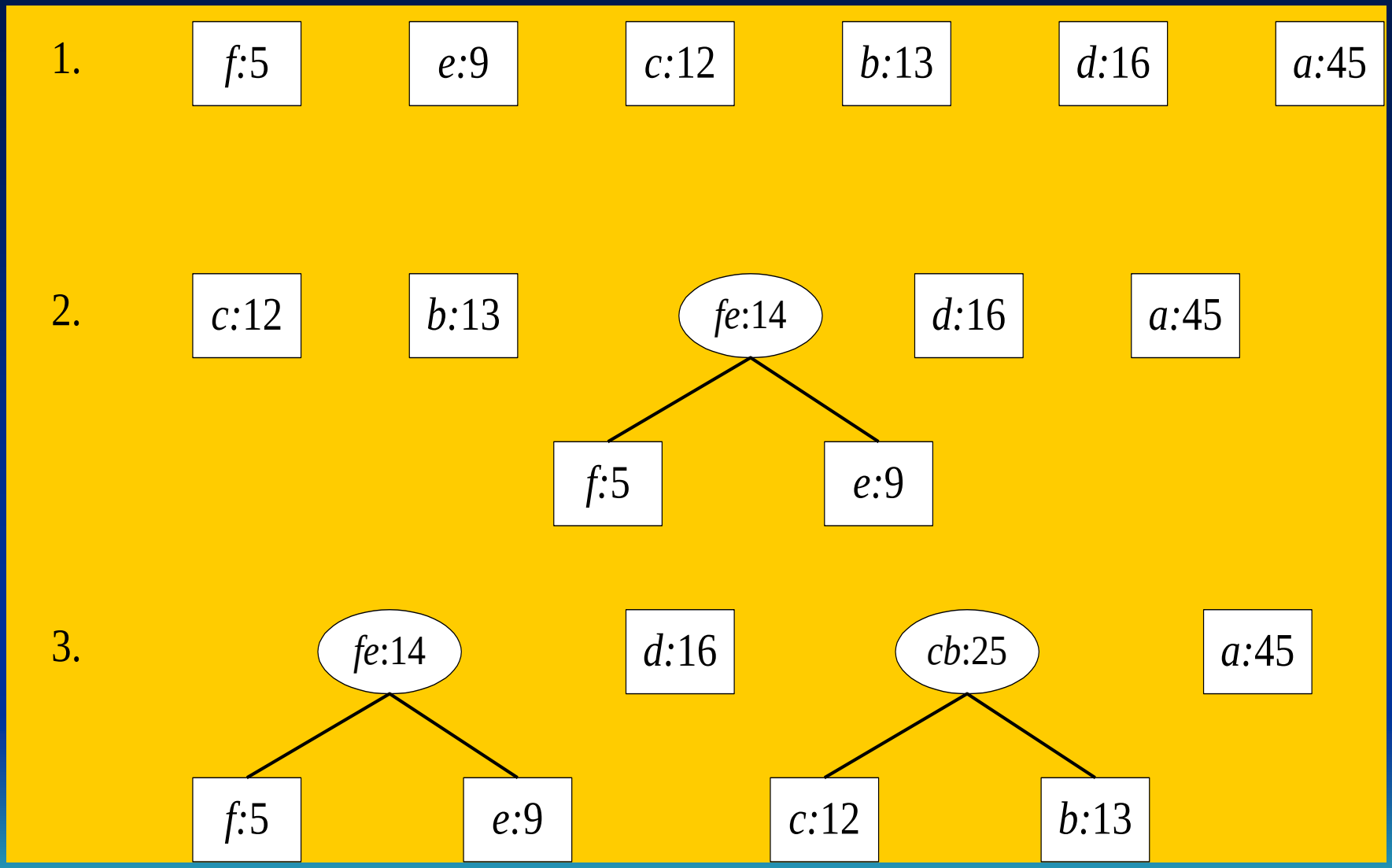
2. Terapkan strategi *greedy* sebagai berikut: pada setiap langkah gabungkan dua buah pohon yang mempunyai frekuensi terkecil pada sebuah akar. Akar mempunyai frekuensi yang merupakan jumlah dari frekuensi dua buah pohon penyusunnya.

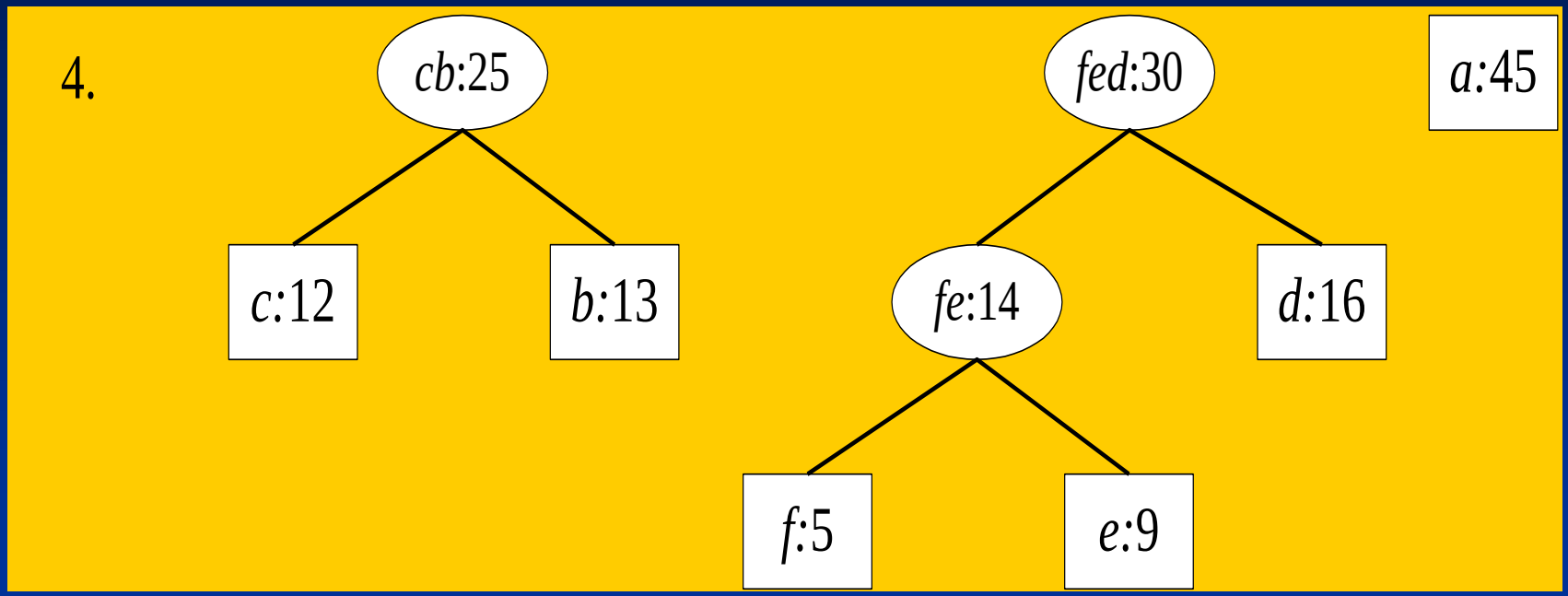
3. Ulangi langkah 2 sampai hanya tersisa satu buah pohon Huffman.

Kompleksitas algoritma Huffman: $O(n \log n)$

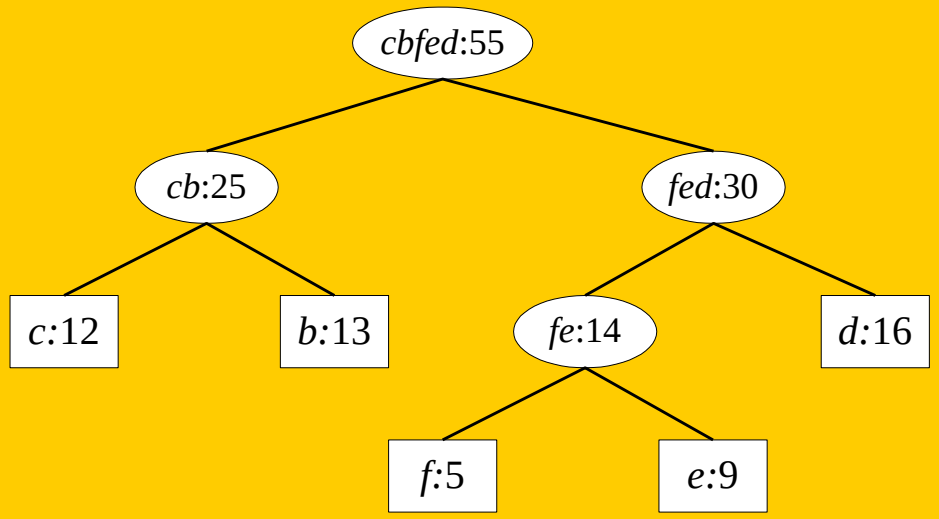
- **Contoh 9.**

Karakter	*a*	*b*	*c*	*d*	*e*	*f*
Frekuensi	45	13	12	16	9	5


1.
f:5
e:9
c:12
b:13
d:16
a:45
2.
c:12
b:13
fe:14
d:16
a:45
f:5
e:9
3.
fe:14
d:16
cb:25
a:45
f:5
e:9
c:12
b:13

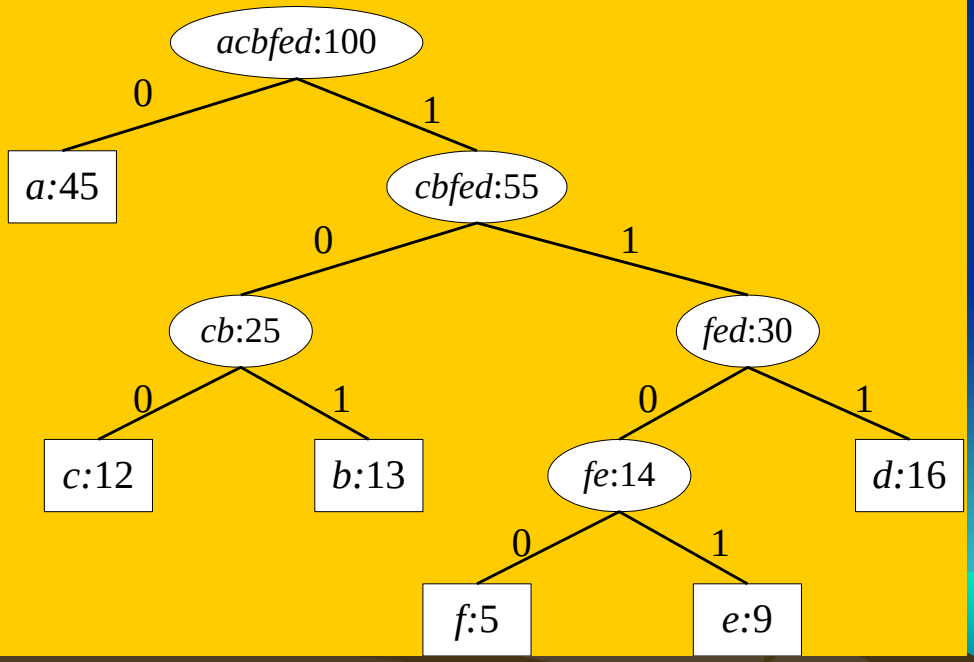

4.
cb:25
fed:30
a:45
c:12
b:13
fe:14
d:16
f:5
e:9

5.


cbfed:55
cb:25
fed:30
c:12
b:13
fe:14
d:16
f:5
e:9

a:45

6


acbfed:100
0
1
a:45
cbfed:55
0
1
cb:25
fed:30
0
1
0
1
c:12
b:13
fe:14
d:16
0
1
f:5
e:9

9. Pecahan Mesir (*Egyptian Fraction*)

Persoalan: Diberikan pecahan *p*/*q*. Dekomposisi pecahan menjadi jumlah dari sejumlah pecahan yang berbeda:

$$\frac{p}{q} = \frac{1}{k_1} + \frac{1}{k_2} + \ldots + \frac{1}{k_n}$$

yang dalam hal ini, $k_1 < k_2 < \ldots < k_n$.

Contoh: $\frac{2}{5} = \frac{1}{3} + \frac{1}{15}$ $\frac{5}{7} = \frac{1}{2} + \frac{1}{5} + \frac{1}{70}$ $\frac{87}{100} = \frac{1}{2} + \frac{1}{5} + \frac{1}{11}$

- Pecahan yang diberikan mungkin mempunyai lebih dari satu representasi Mesir

  Contoh: 8/15 = 1/3 + 1/5

  8/15 = 1/2 + 1/30

- Kita ingin mendekompoisinya dengan jumlah unit pecahan sesedikit mungkin

Strategi *greedy*: pada setiap langkah, tambahkan unit pecahan terbesar ke representasi yang baru terbentuk yang jumlahnya tidak melebihi nilai pecahan yang diberikan

Algoritma:

Input: p/q

1. Mulai dengan $i = 1$
2. Jika $p = 1$, maka $k_i = q$. STOP
3. $1/k_i$ = pecahan terbesar yang lebih kecil dari p/q
4. $p/q = p/q - 1/k_i$
5. Ulangi langkah 2.

- Contoh keluaran:

  8/15 = 1/2 + 1/30

  tetapi, dengan algoritma greedy:

  26/133 = 1/6 + 1/35 + 1/3990 (tidak optimal)

  seharusnya (dengan *brute force*)

  26/133 = 1/7 + 1/19 (optimal)

- Kesimpulan: algoritma *greedy* untuk masalah pecahan Mesir tidak selalu optimal

10. *Connecting wires*

- There are n white dots and n black dots, equally spaced, in a line
- You want to connect each white dot with some one black dot, with a minimum total length of "wire"
- Example:

  - Total wire length above is 1 + 1 + 1 + 5 = 8
  - Do you see a greedy algorithm for doing this?
  - Does the algorithm guarantee an optimal solution?
    - Can you prove it?
    - Can you find a counterexample?

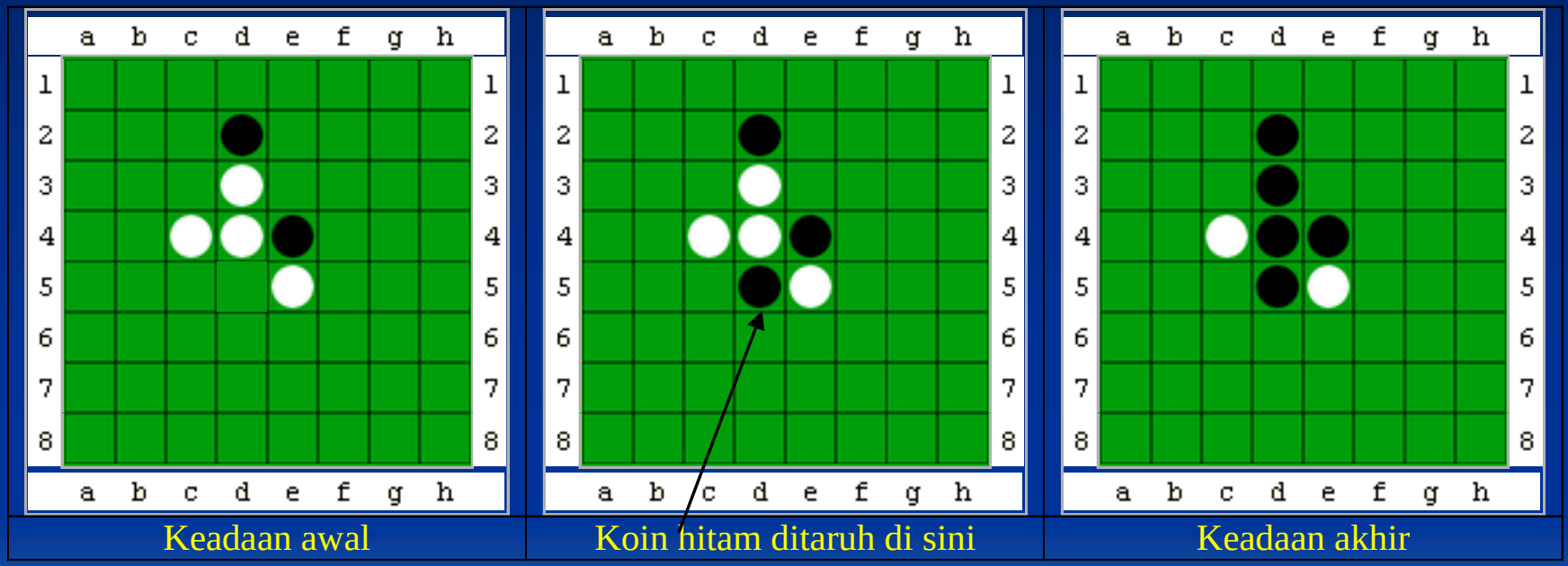
Aplikasi Algoritma *Greedy* pada Permainan Othello (Riversi)

Othello

- *Othello* atau *Reversi* adalah permainan yang menggunakan papan (*board game*) dan sejumlah koin yang berwarna gelap (misalnya hitam) dan terang (misalnya putih).
- Ukuran papan biasanya 8 x 8 kotak (grid) dan jumlah koin gelap dan koin terang masing-masing sebanyak 64 buah. Sisi setiap koin memiliki warna yang berbeda (sisi pertama gelap dan sisi kedua terang).
- Pada permainan ini kita asumsikan warna hitam dan putih. Jumlah pemain 2 orang.

- Dalam permainan ini setiap pemain berusaha mengganti warna koin lawan dengan warna koin miliknya (misalnya dengan membalikkan koin lawan) dengan cara “menjepit” atau memblok koin lawan secara vertikal, horizontal, atau diagonal.

- Barisan koin lawan yang terletak dalam satu garis lurus yang diapit oleh sepasang koin pemain yang *current* diubah (*reverse*) warnanya menjadi warna pemain yang *current*.


a b c d e f g h
1 2 3 4 5 6 7 8
Keadaan awal
Koin hitam ditaruh di sini
Keadaan akhir

- Setiap pemain bergantian meletakkan koinnya. Jika seorang pemain tidak dapat meletakan koin karena tidak ada posisi yang dibolehkan, permainan kembali ke pemain lainnya.

- Jika kedua pemain tidak bisa lagi meletakkan koin, maka permainan berakhir. Hal ini terjadi jika seluruh kotak telah terisi, atau ketika seorang pemain tidak memiliki koin lagi, atau ketika kedua pemain tidak dapat melakukan penempatan koin lagi.

- Pemenangnya adalah pemain yang memiliki koin paling banyak di atas papan.

- Algoritma Greedy dapat diaplikasikan untuk memenangkan permainan.

- Algoritma *greedy* berisi sejumlah langkah untuk melakukan penempatan koin yang menghasilkan jumlah koin maksimal pada akhir permainan.

- Algoritma *Greedy* dipakai oleh komputer pada tipe permainan komputer vs manusia.

Dua strategi greedy heuristik:

1. *Greedy by* jumlah koin

   Pada setiap langkah, koin pemain menuju koordinat yang menghasilkan sebanyak mungkin koin lawan. Strategi ini berusaha memaksimalkan jumlah koin pada akhir permainan dengan menghasilkan sebanyak-banyaknya koin lawan pada setiap langkah.

2. *Greedy by* jarak ke tepi

   Pada setiap langkah, koin pemain menuju ke koordinat yang semakin dekat dengan tepi arena permainan. Strategi ini berusaha memaksimumkan jumlah koin pada akhir permainan dengan menguasai daerah tepi yang sulit untuk dilangkahi koin lawan. Bahkan untuk pojok area yang sulit dilangkahi lawan.

Greedy by Jumlah Koin

1. Himpunan kandidat

   Langkah-langkah yang menghasilkan jumlah koin yang diapit.

2. Himpunan solusi

   Langkah-langkah dari Himpunan kandidat yang memiliki jumlah koin diapit paling besar.

3. Fungsi seleksi

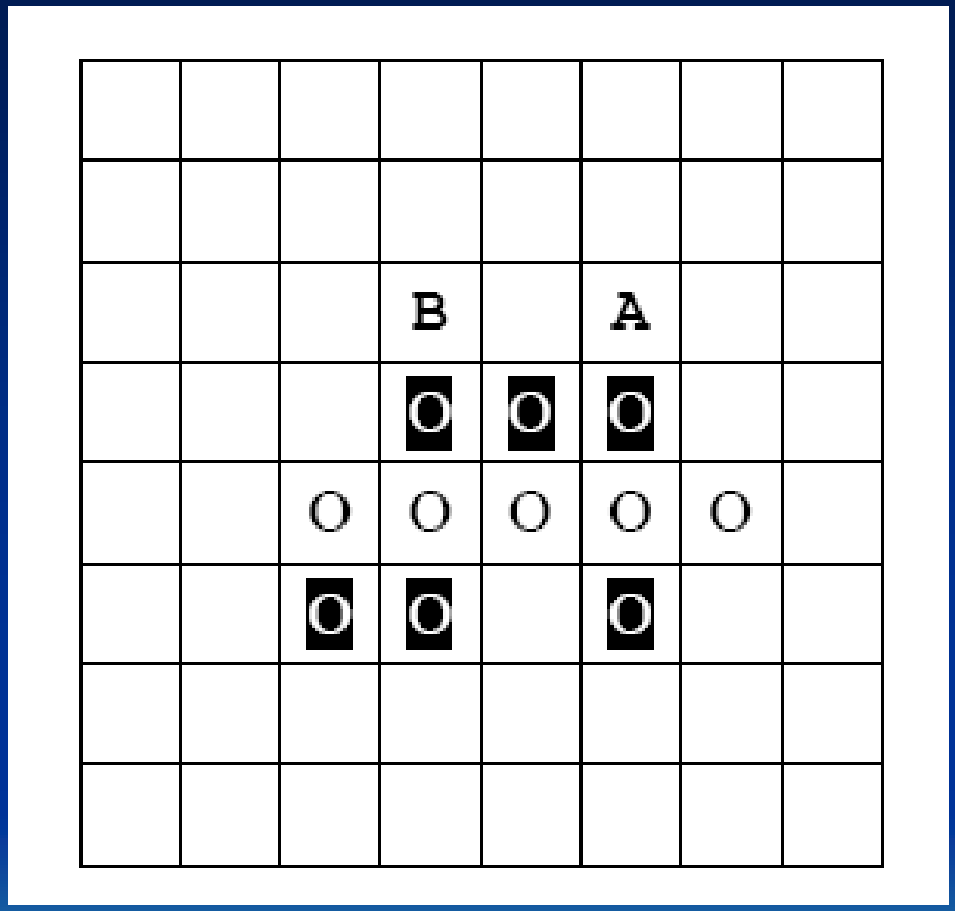
   Pilih langkah yang memiliki jumlah koin diapit paling besar

4. Fungsi kelayakan

   Semua langkah adalah layak

5. Fungsi obyektif

   Maksimumkan jumlah koin lawan


B
A