Bahan Kuliah IF2211 Strategi Algoritma

Algoritma *Greedy*

(Bagian 1)

Oleh: Rinaldi Munir

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika ITB
2025

Pendahuluan

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

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

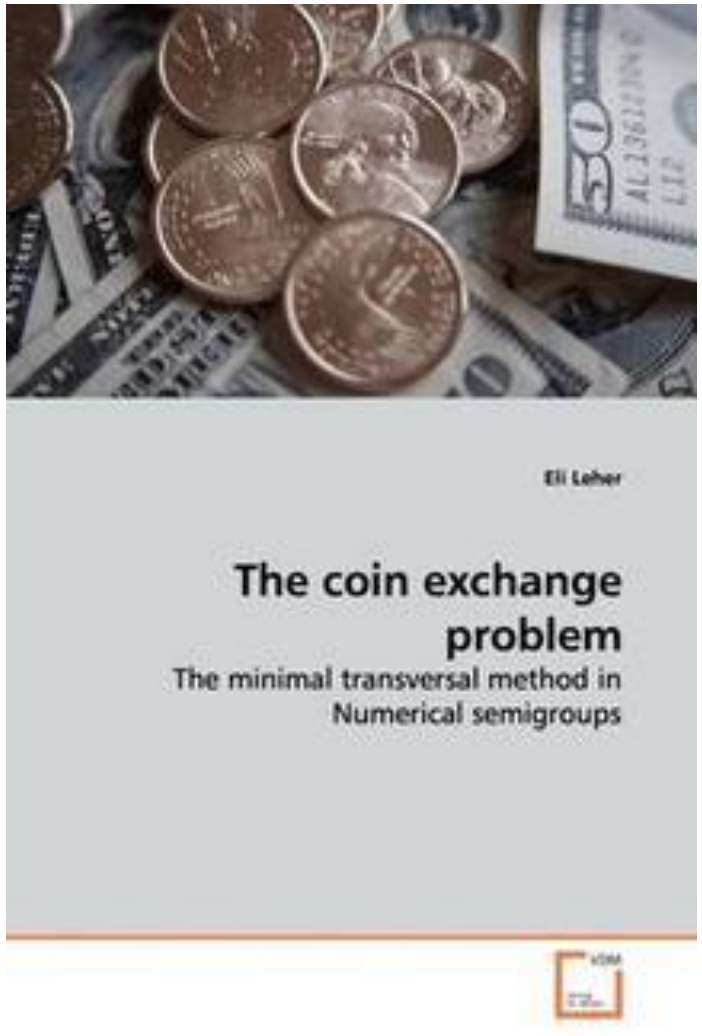
- Hanya ada dua macam persoalan optimasi:
  1. Maksimasi (*maximization*)
  2. Minimasi (*minimization*)

Contoh persoalan optimasi:

Persoalan Penukaran Uang

(*Coin exchange problem*):

Diberikan uang senilai *A*. Tersedia uang koin-koin dalam jumlah yang banyak. 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

- Solusi 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. Tidak bisa mundur lagi (kembali) ke langkah sebelumnya.
- Jadi pada setiap langkah, kita memilih **optimum lokal** (*local optimum*)
- dengan harapan bahwa langkah sisanya mengarah ke solusi **optimum global** (*global optimum*).

Definisi

- **Algoritma *greedy*** adalah algoritma yang memecahkan persoalan secara langkah per langkah (*step by step*) sedemikian sehingga,

  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. dan "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 (dalam jumlah banyak)
  Langkah 1: pilih 1 buah koin 25 (Total = 25)
  Langkah 2: pilih 1 buah koin 5 (Total = 25 + 5 = 30)
  Langkah 3: pilih 1 buah koin 1 (Total = 25+5+1 = 31)
  Langkah 4: pilih 1 buah koin 1 (Total = 25+5+1+1 = 32)

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

- Elemen-elemen algoritma *greedy*:

  1. Himpunan kandidat, *C* : berisi kandidat yang akan dipilih pada setiap Langkah

     (misal: simpul/sisi di dalam graf, job, task, koin, benda, karakter, dsb)

  2. Himpunan solusi, *S* : berisi kandidat yang sudah dipilih

  3. Fungsi solusi: menentukan apakah himpunan kandidat yang dipilih sudah memberikan solusi

  4. Fungsi seleksi (*selection function*): memilih kandidat berdasarkan strategi *greedy* tertentu. Strategi *greedy* ini bersifat heuristik.

  5. Fungsi kelayakan (*feasible*): memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak)

  6. Fungsi obyektif : memaksimumkan atau meminimumkan

- Dengan menggunakan elemen-elemen di atas, maka dapat dikatakan bahwa:

  Algoritma *greedy* melibatkan pencarian sebuah himpunan bagian, *S*, dari himpunan kandidat, *C*; yang dalam hal ini, *S* harus memenuhi beberapa kriteria yang ditentukan, yaitu *S* menyatakan suatu solusi dan *S* dioptimisasi oleh fungsi obyektif.

Tinjau kembali persoalan penukaran uang. Pada persoalan penukaran uang:

- *Himpunan kandidat*: himpunan koin atau pecahan uang yang merepresentasikan nilai 1, 5, 10, 25, paling sedikit mengandung satu koin untuk setiap nilai.
- *Himpunan solusi*: koin-koin yang terpilih.
- *Fungsi solusi*: memeriksa apakah total nilai koin yang dipilih tepat sama jumlahnya dengan nilai uang yang ditukarkan.
- *Fungsi seleksi*: pilihlah koin yang bernilai tertinggi dari himpunan koin yang tersisa.
- *Fungsi kelayakan*: memeriksa apakah koin yang baru dipilih apabila dijumlahkan dengan semua koin yang sudah berada di dalam himpunan tidak melebihi jumlah uang yang harus dibayar.
- *Fungsi obyektif*: jumlah koin yang digunakan minimum.

Skema umum algoritma *greedy*:

```
function greedy(C : himpunan_kandidat) → himpunan_solusi
{ Mengembalikan solusi dari persoalan optimasi dengan algoritma greedy  }
Deklarasi
   x : kandidat
   S : himpunan_solusi

Algoritma:
    S ← {}      { inisialisasi S dengan kosong }
    while (not SOLUSI(S)) and (C ≠ {} ) do
        x ← SELEKSI(C)     { pilih sebuah kandidat dari C}
        C ← C – {x}        { buang x dari C karena sudah dipilih }
        if  LAYAK(S ∪ {x}) then       { x memenuhi kelayakan untuk dimasukkan ke dalam himpunan solusi }
            S ← S ∪ {x}       { masukkan x ke dalam himpunan solusi }
        endif
    endwhile
    {SOLUSI(S) or C = {} }

    if  SOLUSI(S)  then     { solusi sudah lengkap }
       return S
    else
      write('tidak ada solusi')
    endif
```

- Pada akhir setiap lelaran (iterasi), solusi yang terbentuk adalah optimum lokal.
- Pada akhir kalang **while-do** diperoleh optimum global (jika ada).

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

- Alasan:

  1. Algoritma *greedy* tidak beroperasi secara menyeluruh
     terhadap semua kemungkinan 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 persoalan, algoritma *greedy* tidak selalu berhasil memberikan solusi yang optimal, namun sub-optimal. Perhatikan Contoh 2 berikut.

- **Contoh 2**: Tinjau persoalan penukaran uang berikut:

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

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

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

Catatan: Untuk sistem mata uang dollar AS, euro Eropa, dan *crown* Swedia, rupiah Indonesia, algoritma *greedy* selalu memberikan solusi optimum untuk persoalan penukaran uang.

- **Contoh 3**: Uang Rp178.400 dapat dinyatakan dalam jumlah pecahan yang minimal sebagai berikut:
  - Satu lembar uang kertas Rp100.000
  - Satu lembar uang kertas Rp50.000
  - Satu lembar uang kertas Rp20.000
  - Satu lembar uang kertas Rp5.000
  - Satu lembar uang kertas Rp2.000
  - Satu koin uang Rp1000
  - Dua koin uang Rp200

100.000 + 50.000 + 20.000 + 5.000 + 2.000 + 1000 + 2x200 = Rp178.400

- Jika solusi terbaik mutlak tidak terlalu diperlukan, maka algoritma *greedy* dapat digunakan untuk menghasilkan solusi hampiran (*approximation*),

- daripada menggunakan algoritma yang kebutuhan waktunya eksponensial untuk menghasilkan solusi yang eksak.

- Misalnya mencari tur dengan bobot minimal pada persoalan TSP untuk jumlah simpul (*n*) yang banyak dengan algoritma *brute force* dibutuhkan waktu komputasi yang lama untuk menemukannya.

- Dengan algoritma *greedy*, meskipun tur dengan berbobot minimal tidak dapat ditemukan, namun solusi dengan algoritma *greedy* dianggap sebagai hampiran solusi optimal.

- Namun bila algoritma *greedy* dapat menghasilkan solusi optimal, maka keoptimalannya itu harus dapat dibuktikan secara matematis.

- Membuktikan optimalitas algoritma greedy secara matematis adalah tantangan tersendiri.

- Lebih mudah memperlihatkan algoritma *greedy* tidak selalu optimal dengan menunjukkan *counterexample* (contoh kasus yang menunjukkan solusi yang diperoleh tidak optimal),

- misalnya pada persoalan penukaran uang di atas (kadang-kadang solusinya optimal kadang-kadang tidak).

Contoh-contoh persoalan yang diselesaikan dengan Algoritma Greedy

1. Persoalan penukaran uang (*coin exchange problem*)
2. Persoalan memilih aktivitas (*activity selection problem*)
3. Minimisasi waktu di dalam sistem
4. Persoalan *knapsack* (*knapsack problem*)
5. Penjadwalan *job* dengan tenggat waktu (*job schedulling with deadlines*)
6. Pohon merentang minimum (*minimum spanning tree*)
7. Lintasan terpendek (*shortest path*)
8. Kode Huffman (*Huffman code*)
9. Pecahan Mesir (*Egyptian fraction*)

1. Persoalan Penukaran Uang

- Persoalan ini sudah kita bahas sebelumnya
- Di sini kita formulasikan kembali persoalan ini secara matematis
- Persoalan 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*

- Karena $X = \{x_1, x_2, \ldots, x_n\}$ dan x_i bernilai 1 atau 0, maka terdapat 2^n kemungkinan solusi (yaitu terdapat 2^n kemungkinan vektor X)

- Untuk mengevaluasi fungsi obyektif kebutuhan waktunya $O(n)$

- 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(C : himpunan_koin, A : integer) → himpunan_solusi
{ mengembalikan koin-koin yang total nilainya = A, tetapi jumlah koinnya minimum }
Deklarasi
  S : himpunan_solusi
  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
```

- Memilih koin bernilai terbesar dari himpunan dengan *n* koin kompleksitasnya adalah O(*n*), menjumlahkan semua koin di dalam S juga O(*n*).
- Kalang **while** diulang maksimal sebanyak jumlah koin di dalam himpunan, yaitu *n* kali.
- Sehingga kompleksitas algoritma *CoinExchange* adalah $O(n^2)$

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

- Jika himpunan koin sudah terurut menurun, maka kompleksitas algoritma *CoinExchange* adalah *O*(*n*). Namun jika kompleksitas algoritma pengurutan diperhitungkan, maka kompleksitas algoritma *CoinExchange* adalah $O(n^2)$ jika algoritma pengurutan yang digunakan adalah algoritma *brute force*.

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

2. Persoalan memilih aktivitas (*activity selection problem*)

- **Persoalan**: Misalkan terdapat n buah aktivitas, $S = \{1, 2, \ldots, n\}$, yang ingin menggunakan sebuah *resource* (misalnya ruang pertemuan, studio, prosesor, dll). *Resource* hanya dapat digunakan oleh satu aktivitas dalam satu waktu. Setiap kali suatu aktivitas memakai (*occupy*) *resource*, maka aktivitas lain tidak dapat menggunakannya sebelum aktivitas tersebut selesa.

  Setiap aktivitas *i* memiliki waktu mulai s_i dan waktu selesai f_i, yang dalam hal ini $s_i \leq f_i$. Dua aktivitas *i* dan *j* dikatakan **kompatibel** jika interval $[s_i, f_i]$ dan $[s_j, f_j]$ tidak beririsan.

  Persoalan *activity selection problem* ialah bagaimana memilih sebanyak mungkin aktivitas yang dapat dilayani oleh *resource*.

Contoh 4: Contoh instansiasi persoalan. Misalkan terdapat n = 11 buah aktivitas dengan waktu mulai dan waktu selesai masing-masing sebagai berikut:

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*

Langkah-langkah:

- 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 salah satu kandidat solusinya.
- Pilih kandidat solusi yang memiliki jumlah aktivitas paling banyak.
- Kompleksitas waktu algoritmanya $O(n \cdot 2^n)$. Mengapa?

Penyelesaian dengan Algoritma Greedy

- Apa strategi *greedy-nya*?

1. Urutkan semua aktivitas berdasarkan waktu selesainya terurut dari kecil ke besar

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

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

Solusi:

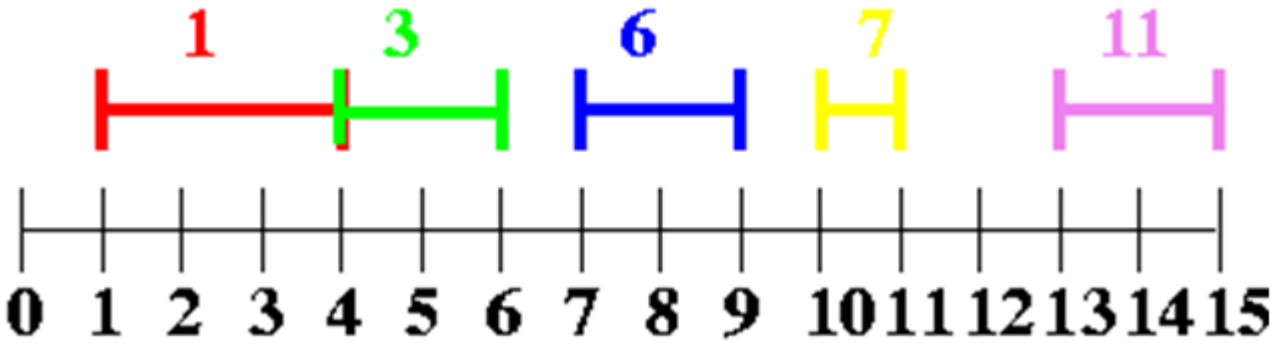
A = {1}, f_j = 4

A = {1, 3}, f_j = 6

A = {1, 3, 6}, f_j = 9

A = {1, 3, 6, 7}, f_j = 11

A = {1, 3, 6, 7, 11}, f_j = 15



Solusi: aktivitas yang dipilih adalah 1, 3, 6, 7, dan 11 (5 aktivitas)

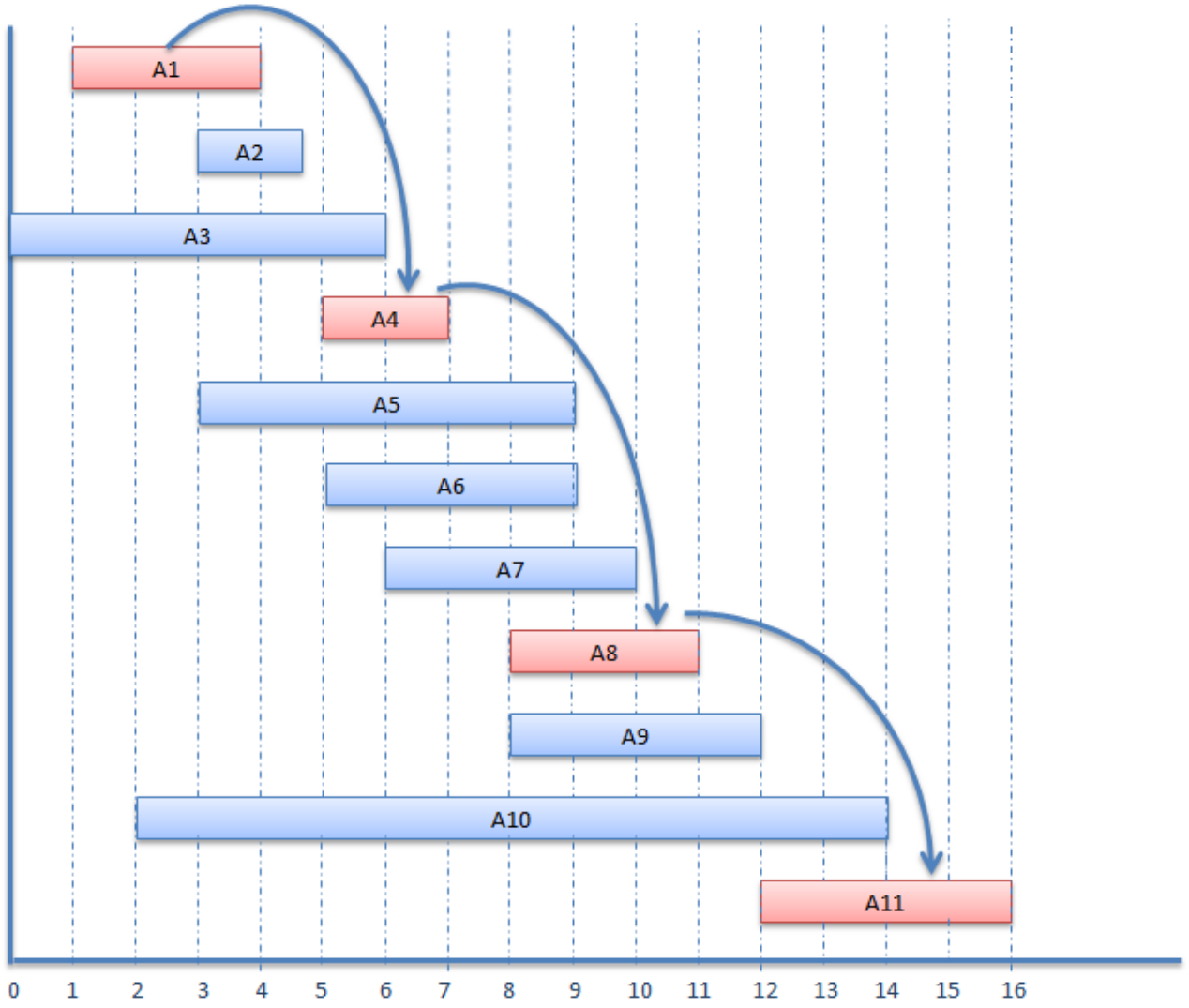
Contoh 5: Contoh lain untuk n = 12 aktivitas

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 *greedy* untuk *activity selection problem*

function *Greedy-Activity-Selector*(s_1, s_2, …, s_n : **integer**, f_1, f_2, …, f_n : **integer**) → **set of integer**
{ Asumsi: aktivitas sudah diurut terlebih dahulu berdasarkan waktu selesai: f1 <= f2 <= ... <= fn }
Deklarasi
i, *j*, *n* : **integer**
A : **set of integer**
Algoritma:
n ← *length*(*s*)
A ← {1} *{ aktivitas nomor 1 selalu terpilih }*
j ← 1
for *i* ← 2 **to** *n* **do**
if $s_i \geq f_j$ **then**
A ← *A* ∪ {*i*]
j ← *i*
endif
endif

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

Pembuktian optimalitas algoritma greedy untuk persoalan *activity selection problem*

- Misalkan S = {1, 2,…, n} adalah himpunan aktivitas yang sudah diurutkan berdasarkan waktu selesai. Pilihan yang tepat adalah selalu memilih aktivitas 1. Mengapa aktivitas 1 selalu memberikan salah satu solusi optimal?
- Asumsikan $A \subseteq S$ adalah solusi optimal, juga diurutkan berdasarkan waktu selesai dan aktivitas pertama di dalam A adalah "k"
- Jika k = 1, maka A dimulai dengan pilihan *greedy*
- Jika $k \neq 1$, maka A tidak dimulai dengan pilihan *greedy*.
- Misalkan solusi optimal lain, sebut B, dengan ukuran yang sama dengan A, yang dimulai dengan aktivitas 1 sebagai aktivitas pertama. Kita dapat menunjukkan bahwa $B = (A - \{ k\}) \cup \{1\}$
- Karena $f_1 \leq f_k$, dan aktivitas-aktivitas di dalam A kompatibel, aktivitas-aktivitas di dalam B juga kompatibel. Karena $|A| = |B|$, maka B juga optimal.
- Karena itu kita menyimpulkan selalu terdapat solusi optimal yang dimulai dengan pilihan *greedy*.

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

$\rightarrow$Boleh juga, tetapi apakah strategi tersbut dijamin memberikan solusi optimal?

$\rightarrow$Jika tidak bisa membuktikan, berikan contoh c*ounterexample* bahwa strategi di atas tidak selalu optimal.

Solusi dengan strategi *greedy* yang kedua: memilih aktivitas yang durasinya paling kecil lebih dahulu

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 yang dipilih adalah 7, 2, 4, 6, dan 11 (5 aktivitas) → pada comtoh ini optimal

- *Counterexample*: memilih berdasarkan durasi terkecil lenboh dahulu

i	s_i	f_i	durasi
1	3	5	2
2	6	8	2
3	1	4	3
4	4	7	3
5	7	10	3

Solusi: aktivitas yang dipilih adalah 1 dan 2 (hanya 2 aktivitas) → tidak optimal

Solusi optimalnya adalah 3 aktivitas (aktivitas 3, 4, dan 5)

3. Minimisasi waktu di dalam sistem

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

  Bagaimana cara meminimumkan total waktu di dalam sistem:

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

- Persoalan ini ekivalen dengan meminimumkan waktu rata-rata pelanggan di dalam sistem.

Contoh 6: Tiga pelanggan dengan

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

Enam urutan pelayanan pelanggan yang mungkin adalah:

Urutan pelanggan	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

- Contoh persoalan lain yang mirip dengan miniminasi waktu di dalam sistem adalah *optimal storage on tapes* (penyimpanan program secara optimal di dalam pita) atau penyimpanan lagu (musik) di dalam pita kaset.

- Pita kaset adalah media penyimpanan sinyal analog pada masa lalu. Pita kaset merupakan media penyimpanan yang bersifat sekuensial.

- Lagu-lagu (atau program-program) disimpan di dalam pita kaset secara sekuensial. Panjang setiap lagu *i* adalah t_i (dalam satuan menit atau detik). Untuk menemukan (*retrieve*) dan memainkan sebuah lagu, pita pada mulanya diposisikan pada bagian awal.

- Jika lagu-lagu di dalam pita disimpan dalam urutan $I = i_1, i_2, \ldots, i_n$, maka lama waktu yang dibutuhkan untuk memainkan lagu i_j sampai selesai adalah $\sum_{i \le k \le j} t_{i_k}$

- Jika semua lagu sering dimainkan, maka waktu rata-rata retrieval (*mean retrieval time* atau MRT) adalah $\frac{1}{n} \sum_{i \le j \le n} t_j$.

- Pada persoalan *optimal storage on tapes*, kita diminta menemukan permutasi *n* buah lagu sedemikian sehingga bila lagu-lagu itu disimpan di dalam pita kaset akan meminimumkan MRT.

- Meminimumumkan MRT ekivalen dengan meminimumkan $d(I) = \sum_{1 \le j \le n} \sum_{1 \le k \le j} t_{i_k}$

Penyelesaian dengan *Exhaustive Search*

- Perhatikan bahwa urutan pelanggan 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*(*C* : himpunan_pelanggan) → himpunan_pelanggan
{ mengembalikan urutan jadwal pelayanan pelanggan yang meminimumkan waktu di dalam sistem }

Deklarasi
S : himpunan_pelanggan
i : *pelanggan*

Algoritma
$S \leftarrow \{\}$
while ($C \neq \{\}$) **do**
$i \leftarrow$ pelanggan yang mempunyai $t[i]$ terkecil
$C \leftarrow C - \{i\}$
$S \leftarrow S \cup \{i\}$
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. Pelanggan sudah terurut berdasarkan
  t_i yang menaik
  Luaran: 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(i)
 endfor
```

- Algoritma *greedy* untuk penjadwalan pelanggan berdasarkan waktu layanan yang terurut menaik akan selalu menghasilkan solusi optimal .

- Hal ini dinyatakan di dalam teorema berikut:

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.

Bukti untuk teorema ini dapat dibaca pada *slide* berikut (Sumber: Ellis Horrowitz & Sartaj Sahni, *Computer Algorithms, 1998)*

Proof: Let $I = i_1, i_2, \ldots, i_n$ be any permutation of the index set $\{1, 2, \ldots, n\}$. Then

$$d(I) = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j} = \sum_{k=1}^{n} (n - k + 1) t_{i_k}$$

If there exist a and b such that $a < b$ and $t_{i_a} > t_{i_b}$, then interchanging i_a and i_b results in a permutation I' with

$$d(I') = \left[\sum_{\substack{k \\ k \neq a \\ k \neq b}} (n - k + 1) t_{i_k} \right] + (n - a + 1) t_{i_b} + (n - b + 1) t_{i_a}$$

Subtracting $d(I')$ from $d(I)$, we obtain

$$\begin{aligned} d(I) - d(I') &= (n - a + 1)(t_{i_a} - t_{i_b}) + (n - b + 1)(t_{i_b} - t_{i_a}) \\ &= (b - a)(t_{i_a} - t_{i_b}) \\ &> 0 \end{aligned}$$

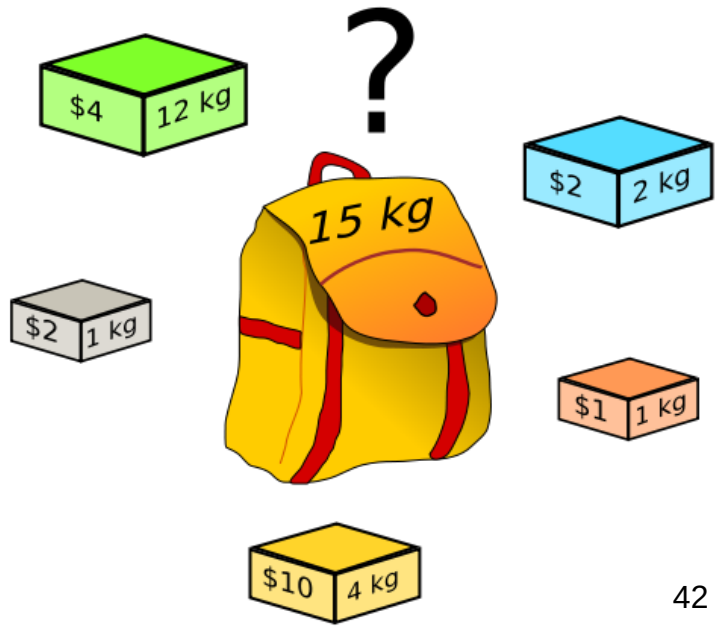
Hence, no permutation that is not in nondecreasing order of the t_i's can have minimum d. It is easy to see that all permutations in nondecreasing order of the t_i's have the same d value. Hence, the ordering defined by $i_j = j, 1 \leq j \leq n$, minimizes the d value. □

(Sumber: Ellis Horrowitz & Sartaj Sahni, *Computer Algorithms, 1998*)

4. Integer 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)

- Formulasi matematis persoalan *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, $\quad i = 1, 2, \ldots, n$

Penyelesaian dengan algoritma *brute force* (*exhaustive search*)

- Sudah dijelaskan pada Bab Algoritma Brute Force (pembahasan *exhaustive search*)

- Kompleksitas algoritma *exhaustive search* untuk persoalan *integer knapsack* adalah $O(n \cdot 2^n)$.

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.

   - Strategi ini 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 <u>tidak menjamin</u> akan memberikan solusi optimal.

Contoh 7. Diberikan 4 buah objek sbb:

$(w_1, p_1) = (6, 12)$; $(w_2, p_2) = (5, 15)$; $(w_3, p_3) = (10, 50)$; $(w_4, p_4) = (5, 10)$;

dan sebuah *knapsack* dengan kapasitas $K = 16$.

Solusi dengan algoritma *greedy*:

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! Apakah untuk kasus lain juga optimal? Perhatikan Contoh 8 berikut.

Contoh 8. Diberikan 6 buah objek sbb:

$(w_1, p_1) = (100, 40)$; $(w_2, p_2) = (50, 35)$; $(w_3, p_3) = (45, 18)$;

$(w_4, p_4) = (20, 4)$; $(w_5, p_5) = (10, 10)$; $(w_6, p_6) = (5, 2)$;

dan sebuah *knapsack* dengan kapasitas $K = 100$. Solusi dengan algoritma *greedy*:

Properti objek				*Greedy by*			Solusi
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*	Optimal
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 persoalan *integer knapsack*.

5. Fractional Knapsack Problem

- *Fractional knapsack problem* merupakan varian persoalan *knapsack* tetapi dengan solusi yang boleh berbentuk pecahan atau fraksi
- Objek yang dimasukkan ke dalam *knapsack* dapat berupa sebagian saja ($0 \leq x_i \leq 1$)

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 pada *integer knapsack* dapat digunakan untuk memilih objek yang akan dimasukkan ke dalam *knapsack*.

- Mari kita bahas satu per satu

Contoh 9. Diberikan 3 buah objek sbb:

$(w_1, p_1) = (18, 25)$; $(w_2, p_2) = (15, 24)$; $(w_3, p_3) = (10, 15)$;

dan sebuah *knapsack* dengan kapasitas $K = 20$.

Solusi dengan algoritma *greedy*:

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 selalu 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 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.

- **Kesimpulan**: Algoritma *greedy* selalu berhasil menemukan solusi optimal untuk persoalan *fractional knapsack* jika strategi memilih objek adalah berdasarkan densitas terbesar lebih dahulu.

- 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 prosedur `FractionalKnapsack`

function *FractionalKnapsack*(*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 + w[i] ≤ K  then
       { masukkan objek i ke dalam knapsack }
       x[i] ← 1
       TotalBobot ← TotalBobot + w[i]
    else   { masukkan fraksi objek tersebut }
       MasihMuatUtuh ← false
       x[i] ← (K – TotalBobot)/w[i]
    endif
  endwhile
  { i > n or not MasihMuatUtuh }

  return  x
```

Kompleksitas waktu algoritma (jika pengurutan tidak diperhitungkan) = $O(n)$.

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

Deskripsi persoalan:

- Ada *n* buah *job* yang akan dikerjakan oleh sebuah mesin;
- Tiap *job* diproses oleh mesin selama 1 satuan waktu dan tenggat waktu (*deadline*) pengerjaan 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?

Contoh 10. Misalkan terdapat 4 *job* (n = 4):

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

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

Andaikan mesin mulai bekerja jam 6.00 pagi, maka itu artinya:

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

- Misalkan *J* adalah himpunan job yang diproses oleh mesin. Maka, 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*.

Penyelesaian dengan *Exhaustive Search*

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

Barisan *job*	Urutan pemrosesan	Total keuntungan (*F*)	Keterangan
{}	-	0	layak
{1}	1	50	layak
{2}	2	10	layak
{3}	3	15	layak
{4}	4	30	layak
{1, 2}	2, 1	60	layak
{1, 3}	1, 3 atau 3, 1	65	layak
{1, 4}	**4, 1**	**80**	layak $\rightarrow$ **optimal**
{2, 3}	2, 3	25	layak
{2, 4}	-	-	tidak layak
{3, 2}	-	-	tidak layak
{3, 4}	4, 3	45	layak
{1, 2, 3}	-	-	tidak layak
{1, 2, 4}	-	-	tidak layak
{1, 3, 4}	-	-	tidak layak
{1, 2, 3, 4}	-	-	tidak layak

Kompleksitas algoritma: $O(n \cdot 2^n)$

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* "

- Misalkan himpunan yang berisi job-job yang sudah dipilih disimpan di dalam himpunan *J*. Tempatkan job *i* di dalam *J* dalam suatu urutan sedemikian sehingga semua job selesai dikerjakan sebelum *deadline*.

Contoh 11: Misalkan terdapat 4 job dengan profit dan *deadline* sebagai berikut:

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

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

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 JobSchedulling(d[1..n] : array of integer, p[1..n] : array of integer) → himpunan_job
{ Menghasilkan barisan job yang akan diproses oleh mesin }

Deklarasi
  i, k : integer
  J : himpunan_job   { solusi }

Algoritma
  J ← {}
  for i ← 1 to  n do
     k ← job yang mempunyai profit terbesar
     if  (semua job di dalam J ∪ {k} layak) then
         J ← J ∪ {k}
    endif
  endfor

  return J
```

Memilih job yang mempunyai p_i terbesar : $O(n)$

Pengulangan while dilakukan sebanyak n kali (sebanyak himpunan job di dalam C)

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

- Agar pencarian solusi lebih sangkil (*effective*), maka job-job diurut terlebih dahulu berdasarkan profitnya dari profit terbesar ke profit kecil.
- Algoritmanya menjadi sebagai berikut:

```
function JobSchedulling2(d[1..n] : array of integer, p[1..n] : array of integer) → himpunan_job
{ Menghasilkan barisan job yang akan diproses oleh mesin
{ Job-job sudah diurutkan terlebih dahulu berdasarkan profit dari besar ke kecil }

Deklarasi
  i : integer
  J : himpunan_job   { solusi }

Algoritma
  J ← {1}                  { job 1 selalu terpilih}
  for i ← 2 to n do
     if (semua job di dalam J ∪ {i} layak) then
        J ← J ∪ {i}
     endif
  endfor
  return J
```

- Algoritma yang lebih detil:

```
1   Algorithm JS(d, j, n)
2   // d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
3   // are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
4   // is the ith job in the optimal solution, 1 ≤ i ≤ k.
5   // Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
6   {
7       d[0] := J[0] := 0; // Initialize.
8       J[1] := 1; // Include job 1.
9       k := 1;
10      for i := 2 to n do
11      {
12          // Consider jobs in nonincreasing order of p[i]. Find
13          // position for i and check feasibility of insertion.
14          r := k;
15          while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
16          if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
17          {
18              // Insert i into J[ ].
19              for q := k to (r + 1) step −1 do J[q + 1] := J[q];
20              J[r + 1] := i; k := k + 1;
21          }
22      }
23      return k;
24  }
```

(Sumber: Ellis Horrowitz & Sartaj Sahni, *Computer Algorithms, 1998)*

Theorem 4.4 The greedy method described above always obtains an optimal solution to the job sequencing problem.

Proof: Let $(p_i, d_i), 1 \leq i \leq n$, define any instance of the job sequencing problem. Let I be the set of jobs selected by the greedy method. Let J be the set of jobs in an optimal solution. We now show that both I and J have the same profit values and so I is also optimal. We can assume $I \neq J$ as otherwise we have nothing to prove. Note that if $J \subset I$, then J cannot be optimal. Also, the case $I \subset J$ is ruled out by the greedy method. So, there exist jobs a and b such that $a \in I$, $a \notin J$, $b \in J$, and $b \notin I$. Let a be a highest-profit job such that $a \in I$ and $a \notin J$. It follows from the greedy method that $p_a \geq p_b$ for all jobs b that are in J but not in I. To see this, note that if $p_b > p_a$, then the greedy method would consider job b before job a and include it into I.

Now, consider feasible schedules S_I and S_J for I and J respectively. Let i be a job such that $i \in I$ and $i \in J$. Let i be scheduled from t to $t+1$ in S_I and t' to $t'+1$ in S_J. If $t < t'$, then we can interchange the job (if any) scheduled in $[t', t'+1]$ in S_I with i. If no job is scheduled in $[t', t'+1]$ in I, then i is moved to $[t', t'+1]$. The resulting schedule is also feasible. If $t' < t$, then a similar transformation can be made in S_J. In this way, we can obtain schedules S'_I and S'_J with the property that all jobs common to I and J are scheduled at the same time. Consider the interval $[t_a, t_a+1]$ in S'_I in which the job a (defined above) is scheduled. Let b be the job (if any) scheduled in S'_J in this interval. From the choice of $a, p_a \geq p_b$. Scheduling a from t_a to t_a+1 in S'_J and discarding job b gives us a feasible schedule for job set $J' = J - \{b\} \cup \{a\}$. Clearly, J' has a profit value no less than that of J and differs from I in one less job than J does.

By repeatedly using the transformation just described, J can be transformed into I with no decrease in profit value. So I must be optimal. □

(Sumber: Ellis Horrowitz & Sartaj Sahni, *Computer Algorithms, 1998)*

Bersambung ke bagian 2