



## MODUL PERKULIAHAN

# TFC251 – Praktikum Struktur Data

## Algoritma Greedy

<b>Penyusun Modul</b>	: Suamanda Ika Novichasari, M.Kom
<b>Minggu/Pertemuan</b>	: 14
<b>Sub-CPMK/Tujuan Pembelajaran</b>	: <ol style="list-style-type: none"><li>1. Mahasiswa mampu menerapkan algoritma greedy pada kasus knapsack menggunakan bahasa pemrograman python</li></ol>
<b>Pokok Bahasan</b>	: <ol style="list-style-type: none"><li>1. Algoritma Greedy</li><li>2. Knapsack</li></ol>

**Program Studi Teknologi Informasi (S1)**  
**Fakultas Teknik**  
**Universitas Tidar**  
**Tahun 2024**



# Materi 11

## ALGORITMA GREEDY

---

Petunjuk Praktikum :

- Cobalah semua contoh penyelesaian kasus dengan kode program yang terdapat pada semua sub bab dalam modul ini menggunakan google colab.
- Dokumentasikan kegiatan dalam bentuk laporan studi kasus sesuai template laporan praktikum yang telah ditentukan.

Masalah Knapsack adalah salah satu masalah optimasi klasik dalam ilmu komputer dan teori keputusan. Masalah Knapsack sering digunakan dalam konteks seperti alokasi sumber daya, pemilihan portofolio investasi, dan pengemasan barang. Algoritma dinamis dan teknik pemrograman lainnya digunakan untuk menemukan solusi optimal atau mendekati optimal untuk masalah ini, menjadikannya topik yang menarik dan penting dalam studi algoritma dan optimasi.

### 10.1 Algoritma Greedy

Algoritma greedy adalah pendekatan untuk memecahkan masalah dengan membuat keputusan secara bertahap, memilih opsi terbaik yang tersedia pada setiap langkah tanpa mempertimbangkan dampak jangka panjang. Inti dari algoritma greedy adalah selalu memilih pilihan yang tampak paling optimal pada saat itu. Karakteristik Utama dari algoritma greedy adalah sebagai berikut :

- Pemilihan Lokal: Setiap keputusan didasarkan pada informasi lokal, tanpa memperhatikan keseluruhan struktur masalah.
- Solusi Optimal Lokal: Algoritma selalu memilih solusi yang tampak paling optimal secara lokal di setiap langkah.
- Kepastian: Algoritma greedy tidak selalu menjamin solusi optimal secara global untuk semua jenis masalah, tetapi dalam kasus tertentu (misalnya, masalah dengan properti optimal substruktur), algoritma ini dapat memberikan solusi optimal global.

Berikut merupakan contoh sederhana penggunaan algoritma greedy untuk menentukan cara terbaik memberikan kembalian dalam jumlah minimum koin dari sekumpulan denominasi yang sudah ditentukan.

```
def coin_change_greedy(n):
    coins = [20, 10, 5, 1]
    i = 0

    while(n>0):
        if(coins[i] > n):
            i = i+1
        else:
            print(coins[i])
            n = n-coins[i];
            print("\n\n")
```

Penjelasan langkah demi langkahnya adalah sebagai berikut :

- Definisi Fungsi coin\_change\_greedy(n):
  - Parameter n adalah jumlah uang yang akan dikembalikan.
  - Daftar coins berisi denominasi koin yang tersedia, diurutkan dari nilai terbesar ke terkecil: [20, 10, 5, 1].
  - Variabel i diinisialisasi ke 0, yang akan digunakan untuk mengakses elemen dalam daftar coins.
- Proses dalam while Loop:
  - Kondisi while(n > 0): Loop ini akan terus berjalan selama nilai n masih lebih besar dari 0.
  - Kondisi if(coins[i] > n): Jika koin saat ini (dengan indeks i) lebih besar dari nilai n, indeks i ditingkatkan untuk mencoba koin berikutnya yang lebih kecil.
  - Kondisi else: Jika koin saat ini tidak lebih besar dari n, koin tersebut dicetak dan nilai n dikurangi dengan nilai koin tersebut (n = n - coins[i]).

Jika fungsi tersebut dipanggil dengan memberikan nilai 57 sebagai parameter masukan maka pecahan yang dihasilkan adalah 20,20,10,5,1,1 seperti berikut:

```
coin_change_greedy(57)
20
20
10
5
1
1
```

Algoritma Dijkstra yang sudah dibahas pada modul 10 merupakan contoh khusus dari algoritma greedy yang diterapkan pada masalah jarak terpendek dalam graf berbobot non-negatif. Algoritma Dijkstra menggunakan prinsip greedy untuk memastikan bahwa setiap langkah yang diambil adalah optimal secara lokal, yang pada akhirnya memberikan solusi optimal global untuk masalah jarak terpendek pada graf dengan bobot non-negatif. Walaupun tidak semua masalah yang menggunakan algoritma greedy akan memberikan solusi optimal global, algoritma Dijkstra adalah contoh di mana prinsip greedy bekerja dengan baik dan memberikan solusi optimal global.

Contoh masalah lainnya yang bisa diselesaikan dengan Algoritma Greedy antara lain :

- Algoritma Pohon Merentang Minimum Kruskal: Menemukan pohon merentang minimum dalam graf berbobot (Sudah dibahas di modul Graf).
- Pengkodean Huffman: Membuat kode awalan optimal untuk simbol-simbol berdasarkan frekuensi kemunculannya.
- Masalah Ransel Pecahan: Memilih barang-barang paling berharga untuk dimasukkan ke dalam ransel dengan batasan kapasitas berat.
- Masalah Pemilihan Aktivitas: Memilih sebanyak mungkin aktivitas yang tidak saling bertabrakan dari sebuah rangkaian aktivitas.
- Penjadwalan dan Alokasi Sumber Daya : Membuat jadwal pekerjaan atau mengalokasikan sumber daya secara efisien.

Berikut contoh implementasi sederhana masalah pemilihan aktivitas dengan menggunakan prinsip kerja algoritma greedy.

- Diberikan  $n$  aktivitas beserta waktu mulai dan selesainya. Pilih jumlah maksimum aktivitas yang dapat dilakukan oleh satu orang, dengan asumsi bahwa seseorang hanya dapat mengerjakan satu aktivitas dalam satu waktu.

Contoh Implementasi:

Perhatikan 6 kegiatan berikut  
diurutkan berdasarkan waktu selesai.

`mulai[] = {1, 3, 0, 5, 8, 5};`

`selesai[] = {2, 4, 6, 7, 9, 9};`

Seseorang dapat melakukan paling banyak empat aktivitas.

Hasil serangkaian aktivitas maksimal yang dapat dilakukan adalah {0, 1, 3, 4} ( Ini adalah indeks di start[] dan selesai[] )

```

# n --> Jumlah total kegiatan
# s[]--> Array yang berisi waktu mulai semua aktivitas
# f[] --> Array yang berisi waktu selesai semua aktivitas

def printMaxActivities(s, f ):
    n = len(f)
    print ("The following activities are selected")


    # aktivitas pertama selalu terpilih duluan
    i = 0
    print (i),

    # Pertimbangkan aktivitas lainnya
    for j in range(n):

        # Jika aktivitas ini memiliki waktu mulai lebih besar
        # atau sama dengan waktu selesai aktivitas yang dipilih sebelumnya,
        # maka pilih aktivitas tersebut
        if s[j] >= f[i]:
            print (j),
            i = j

#Program untuk cek pemanggilan fungsi di atas
s = [1, 3, 0, 5, 8, 5]
f = [2, 4, 6, 7, 9, 9]
printMaxActivities(s, f)

```

 The following activities are selected  
0  
1  
3  
4

Kelebihan Algoritma Greedy	Kekurangan Algoritma Greedy
<ul style="list-style-type: none"> <li>• Mudah untuk diterapkan dan dipahami.</li> <li>• Efisien untuk Masalah Tertentu</li> <li>• Waktu Eksekusi Cepat</li> <li>• Proses pengambilan keputusan dalam algoritma greedy mudah dipahami dan dijelaskan.</li> <li>• Dapat Digunakan sebagai Komponen Algoritma yang Lebih Kompleks.</li> </ul>	<ul style="list-style-type: none"> <li>• Algoritma greedy memprioritaskan solusi optimal lokal daripada solusi optimal global, sehingga kadang menghasilkan solusi suboptimal.</li> <li>• Sulit Membuktikan Optimalitas</li> <li>• Urutan data masukan bisa mempengaruhi hasil yang dihasilkan oleh algoritma greedy.</li> <li>• Algoritma greedy tidak cocok untuk semua masalah dan mungkin tidak efektif untuk masalah dengan batasan yang kompleks.</li> </ul>

## 10.2 Masalah Knapsack

Masalah knapsack (atau masalah ransel) adalah salah satu masalah optimasi kombinatorial yang paling terkenal. Ini sering digunakan dalam berbagai aplikasi dari ilmu komputer hingga operasi riset. Masalah knapsack memiliki beberapa varian, tetapi umumnya, mereka melibatkan memilih item untuk dimasukkan ke dalam ransel agar memaksimalkan nilai total tanpa melebihi kapasitas berat ransel. Berikut penjelasan rinci tentang berbagai jenis masalah knapsack:

### 1. Masalah Knapsack 0/1

Deskripsi:

- Anda memiliki sejumlah item, masing-masing dengan berat dan nilai tertentu.
- Anda harus memutuskan apakah akan memasukkan setiap item ke dalam ransel atau tidak (0 atau 1).
- Tujuannya adalah memaksimalkan nilai total item dalam ransel tanpa melebihi kapasitas berat ransel.

Pendekatan Penyelesaian:

- Pemrograman Dinamis: Menggunakan tabel untuk menyimpan solusi submasalah.
- Algoritma Greedy: Tidak efektif untuk masalah knapsack 0/1, karena tidak selalu memberikan solusi optimal.

### 2. Masalah Knapsack Pecahan (Fractional Knapsack)

Deskripsi:

- Mirip dengan masalah knapsack 0/1, tetapi Anda dapat memilih untuk mengambil sebagian dari item.
- Tujuannya tetap memaksimalkan nilai total dalam ransel tanpa melebihi kapasitas.

Pendekatan Penyelesaian:

- Algoritma Greedy: Paling efektif untuk masalah knapsack pecahan, karena dapat memberikan solusi optimal dengan memilih item berdasarkan nilai per satuan berat.

### Contoh:

Kapasitas ransel = 50

Barang:

- Barang 1: berat = 10, nilai = 60
- Barang 2: berat = 20, nilai = 100
- Barang 3: berat = 30, nilai = 120

Pilih barang untuk memaksimalkan nilai tanpa melebihi kapasitas ransel.

### Solusi :

1. Knapsack (0/1) dengan pemrograman dinamis :

Menggunakan sebuah Fungsi knapsack yang menerima empat parameter: wt (array berisi berat setiap item), val (array berisi nilai setiap item), W (kapasitas maksimum ransel), dan n (jumlah item yang tersedia). Program berikut menggunakan Pendekatan rekursif dan memorization.

```
def knapsack(wt, val, W, n):  
  
    # base conditions  
    if n == 0 or W == 0:  
        return 0  
    if t[n][W] != -1:  
        return t[n][W]  
  
    # choice diagram code  
    if wt[n-1] <= W:  
        t[n][W] = max(  
            val[n-1] + knapsack(  
                wt, val, W-wt[n-1], n-1),  
            knapsack(wt, val, W, n-1))  
        return t[n][W]  
    elif wt[n-1] > W:  
        t[n][W] = knapsack(wt, val, W, n-1)  
        return t[n][W]  
  
    # Driver code  
    if __name__ == '__main__':  
        profit = [60, 100, 120]  
        weight = [10, 20, 30]  
        W = 50  
        n = len(profit)  
  
        # We initialize the matrix with -1 at first.  
        t = [[-1 for i in range(W + 1)] for j in range(n + 1)]  
        print("Maximum value that can be obtained is",knapsack(weight, profit, W, n))  
  
Maximum Value that can be obtained is 220
```

Bagian rekursif dari program tersebut terletak pada fungsi knapsack(wt, val, W, n), di mana algoritma melakukan rekursi untuk memecahkan masalah knapsack menjadi submasalah yang lebih kecil. Pada bagian ini, terdapat pemanggilan fungsi knapsack secara rekursif untuk kedua pilihan: ketika item ke- $n$  dimasukkan ke dalam ransel dan ketika item ke- $n$  tidak dimasukkan ke dalam ransel.

Bagian memoisasi dari program tersebut adalah penggunaan matriks  $t$  untuk menyimpan hasil perhitungan dari submasalah yang sudah dipecahkan sebelumnya. Sebelum memecahkan suatu submasalah, program memeriksa apakah hasilnya sudah tersedia dalam matriks  $t$ . Jika sudah tersedia, maka hasilnya langsung digunakan, tanpa perlu melakukan perhitungan ulang. Hal ini memungkinkan program untuk menghindari perhitungan ulang yang tidak perlu dan meningkatkan efisiensi secara keseluruhan.

Kompleksitas algoritma knapsack menggunakan pendekatan pemrograman dinamis adalah  $O(n \cdot W)$ , di mana  $n$  adalah jumlah item yang tersedia dan  $W$  adalah kapasitas maksimum ransel. Hal ini disebabkan oleh fakta bahwa algoritma melakukan iterasi untuk setiap item dan untuk setiap kapasitas ransel dari 0 hingga  $W$ , dan untuk setiap iterasi tersebut, hanya ada satu panggilan fungsi rekursi. Dengan menggunakan pendekatan pemrograman dinamis, algoritma dapat menghindari perhitungan ulang solusi submasalah yang sama, sehingga kompleksitasnya dapat dihitung secara efisien.

## 2. Knapsack Pecahan dengan Algoritma Greedy

Pertama, setiap item direpresentasikan sebagai objek dalam kelas Item, yang memiliki atribut berat (weight), nilai (value), dan rasio nilai terhadap berat (ratio). Kemudian, item-item tersebut diurutkan berdasarkan rasio nilai terhadap berat mereka. Selanjutnya, program melakukan iterasi melalui item-item yang telah diurutkan, menghitung fraksi dari item yang dapat dimasukkan ke dalam ransel berdasarkan sisa kapasitas ransel dan berat item, dan memperbarui total nilai maksimum yang diperoleh serta kapasitas sisa ransel. Setelah iterasi selesai, program mengembalikan total nilai maksimum yang diperoleh dari ransel.



```

class Item:
    def __init__(self, weight, value):
        self.weight = weight
        self.value = value
        # Calculate the value-to-weight ratio for each item
        self.ratio = value / weight

def fractional_knapsack(items, capacity):
    # Sort items by their value-to-weight ratio in non-increasing order
    items.sort(key=lambda x: x.ratio, reverse=True)

    total_value = 0

    # Initialize remaining capacity of the knapsack
    remaining_capacity = capacity

    # Iterate through the sorted items list
    for item in items:
        if remaining_capacity <= 0:
            break

        # Calculate fraction of the item that fits into the knapsack
        fraction = min(1, remaining_capacity / item.weight)

        # Update total value and remaining capacity
        total_value += fraction * item.value
        remaining_capacity -= fraction * item.weight

    # Return total maximum value obtained
    return round(total_value, 2)

# Example usage
items = [Item(10, 60), Item(20, 100), Item(30, 120)]
capacity = 50
print("Maximum Value that can be Obtained is", fractional_knapsack(items, capacity))

```

Maximum Value that can be Obtained is 240.0

Pada contoh penggunaan, item-item diinisialisasi dan dimasukkan ke dalam daftar items, kemudian algoritma fractional knapsack dipanggil dengan daftar tersebut dan kapasitas ransel. Hasilnya dicetak.

Kompleksitas algoritma fractional knapsack adalah  $O(n \log n)$ , di mana  $n$  adalah jumlah item yang tersedia. Ini disebabkan oleh proses pengurutan item berdasarkan rasio nilai terhadap beratnya dalam urutan non-meningkat. Setelah pengurutan, algoritma melakukan iterasi melalui item-item tersebut, yang membutuhkan waktu linier  $O(n)$  untuk menentukan fraksi dari setiap item yang akan dimasukkan ke dalam ransel. Oleh karena itu, total kompleksitasnya adalah  $O(n \log n) + O(n) = O(n \log n)$ .

## Latihan 1

---

Jelaskan apa saja perbedaan dari penerapan algoritma greedy untuk pemilihan aktivitas berikut ini dibandingkan dengan yang ada pada halaman 4-5 diatas !

```
data = {
    "start_time": [2, 6, 4, 10, 13, 7],
    "finish_time": [5, 10, 8, 12, 14, 15],
    "activity": ["Homework", "Presentation", "Term paper", "Volleyball practice", "Biology lecture", "Hangout"]
}

selected_activity = []
start_position = 0
# sorting the items in ascending order with respect to finish time
tem = 0
for i in range(0, len(data['finish_time'])):
    for j in range(0, len(data['finish_time'])):
        if data['finish_time'][i] < data['finish_time'][j]:
            tem = data['activity'][i], data['finish_time'][i], data['start_time'][i]
            data['activity'][i], data['finish_time'][i], data['start_time'][i] = data['activity'][j], data['finish_time'][j], data['start_time'][j]
            data['activity'][j], data['finish_time'][j], data['start_time'][j] = tem

# by default, the first activity is inserted in the list of activities to be selected.
selected_activity.append(data['activity'][start_position])
for pos in range(len(data['finish_time'])):
    if data['start_time'][pos] >= data['finish_time'][start_position]:
        selected_activity.append(data['activity'][pos])
        start_position = pos
print(f"The student can work on the following activities: {selected_activity}")
# Results
# The student can work on the following activities: ['Homework', 'Presentation', 'Volleyball practice', 'Biology lecture']
```

## Latihan 2

---

### Masalah Pengepakan Sampah

Diberikan  $n$  item dengan bobot berbeda dan masing-masing wadah berkapasitas  $c$ , masukkan setiap item ke dalam wadah sedemikian rupa sehingga jumlah total wadah yang digunakan dapat diminimalkan. Dapat diasumsikan bahwa semua item memiliki bobot lebih kecil dari kapasitas sampah.

#### Contoh:

- Masukan:
  - $\text{berat[]} = \{4, 8, 1, 4, 2, 1\}$
  - Kapasitas Bin  $c = 10$
- Keluaran: 2
  - dibutuhkan minimal 2 tempat sampah untuk menampung semua barang
- Masukan:
  - $\text{berat[]} = \{9, 8, 2, 2, 5, 4\}$
  - Kapasitas Bin  $c = 10$
- Keluaran: 4
  - Kami membutuhkan minimal 4 tempat sampah untuk menampung semua barang.

## Latihan 3

---

Diberikan array berukuran  $n$  yang memiliki spesifikasi sebagai berikut:

- Setiap elemen dalam array berisi polisi atau pencuri.
- Setiap polisi hanya dapat menangkap satu pencuri.
- Seorang polisi tidak dapat menangkap pencuri yang jaraknya lebih dari  $K$  unit dari polisi tersebut.

Tentukan jumlah maksimum pencuri yang bisa ditangkap !

Contoh:

- Masukan :
  - `arr[] = {'P', 'T', 'T', 'P', 'T'}`,  $k = 1$ .
- Keluaran : 2.
  - Di sini maksimal 2 pencuri yang bisa ditangkap terlebih dahulu polisi menangkap pencuri pertama dan polisi kedua-manusia dapat menangkap pencuri kedua atau ketiga.
- Masukan :
  - `arr[] = {'T', 'T', 'P', 'P', 'T', 'P'}`,  $k = 2$ .
- Keluaran : 3.
- Masukan :
  - `arr[] = {'P', 'T', 'P', 'T', 'T', 'P'}`,  $k = 3$ .
- Keluaran : 3.