Menganalisa Algoritma

```
Procedure BubbleSort (input/output L: TabelInt, input n: integer)
{Mengurutkan table L[1..N] sehingga terurut menaik dengan metode pengurutan bubble sort.
Masukkan: Tabel L yang sudah terdefinisi nilai-nilainya
Keluaran: Tabel L yang terurut menaik sedemekian sehingga
                                        L[1] ≤ L[2] ≤ ⋯ ≤ L[N].
}
Deklarasi
 i        Integer    (pencacah untuk jumlah Langkah)
 k        Integer    (pencacah, untuk penjumlahan pada setiap langkah)
 temp     Integer    (peubah bantu untuk pertukaran)

Algoritma
        For i ← 1 to n-1 do
                For k ← n n downto i+1 do
                        If L[k] =L[k-1] then
                                {pertukaran L[k] dengan L[k-1]}
                                temp ← L[k]
                                L[k] ← L[k − 1]
                                L[k − 1] ← temp
                        endif
                endfor
        endfor
```

Jika ada larik L dengan 10 buah elemen yang berisi angka-angka random:

7	10	16	13	4	12	3	81	75	26
1	2	3	4	5	6	7	8	9	10

Jelaskan bagaimana proses kerja tahapan algoritma di atas sampai dicapai angka yang berurutan

```
Array awal: [7, 10, 16, 13, 4, 12, 3, 81, 75, 26]
Iterasi ke-1
j = 1: [7, 10, 16, 13, 4, 12, 3, 81, 75, 26] (Tidak ada pertukaran)
j = 2: [7, 10, 16, 13, 4, 12, 3, 81, 75, 26] (Tidak ada pertukaran)
j = 3: [7, 10, 13, 16, 4, 12, 3, 81, 75, 26]
j = 3: [7, 10, 13, 16, 4, 12, 3, 81, 75, 26] (Tidak ada pertukaran)
j = 4: [7, 10, 13, 4, 16, 12, 3, 81, 75, 26]
j = 4: [7, 10, 13, 4, 16, 12, 3, 81, 75, 26] (Tidak ada pertukaran)
j = 5: [7, 10, 13, 4, 12, 16, 3, 81, 75, 26]
j = 5: [7, 10, 13, 4, 12, 16, 3, 81, 75, 26] (Tidak ada pertukaran)
j = 6: [7, 10, 13, 4, 12, 3, 16, 81, 75, 26]
j = 6: [7, 10, 13, 4, 12, 3, 16, 81, 75, 26] (Tidak ada pertukaran)
j = 7: [7, 10, 13, 4, 12, 3, 16, 81, 75, 26] (Tidak ada pertukaran)
j = 8: [7, 10, 13, 4, 12, 3, 16, 75, 81, 26]
j = 8: [7, 10, 13, 4, 12, 3, 16, 75, 81, 26] (Tidak ada pertukaran)
j = 9: [7, 10, 13, 4, 12, 3, 16, 75, 26, 81]
j = 9: [7, 10, 13, 4, 12, 3, 16, 75, 26, 81] (Tidak ada pertukaran)
Iterasi ke-2
j = 1: [7, 10, 13, 4, 12, 3, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 2: [7, 10, 13, 4, 12, 3, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 3: [7, 10, 4, 13, 12, 3, 16, 75, 26, 81]
j = 3: [7, 10, 4, 13, 12, 3, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 4: [7, 10, 4, 12, 13, 3, 16, 75, 26, 81]
j = 4: [7, 10, 4, 12, 13, 3, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 5: [7, 10, 4, 12, 3, 13, 16, 75, 26, 81]
j = 5: [7, 10, 4, 12, 3, 13, 16, 75, 26, 81] (Tidak ada pertukaran)
```

```
j = 6: [7, 10, 4, 12, 3, 13, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 7: [7, 10, 4, 12, 3, 13, 16, 75, 26, 81] (Tidak ada pertukaran)
j = 8: [7, 10, 4, 12, 3, 13, 16, 26, 75, 81]
j = 8: [7, 10, 4, 12, 3, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Iterasi ke-3
j = 1: [7, 10, 4, 12, 3, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 2: [7, 4, 10, 12, 3, 13, 16, 26, 75, 81]
j = 2: [7, 4, 10, 12, 3, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 3: [7, 4, 10, 12, 3, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 4: [7, 4, 10, 3, 12, 13, 16, 26, 75, 81]
j = 4: [7, 4, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 5: [7, 4, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 6: [7, 4, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 7: [7, 4, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Iterasi ke-4
j = 1: [4, 7, 10, 3, 12, 13, 16, 26, 75, 81]
j = 1: [4, 7, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 2: [4, 7, 10, 3, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 3: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81]
j = 3: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 4: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 5: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 6: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Iterasi ke-5
j = 1: [4, 7, 3, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 2: [4, 3, 7, 10, 12, 13, 16, 26, 75, 81]
j = 2: [4, 3, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 3: [4, 3, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 4: [4, 3, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 5: [4, 3, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Iterasi ke-6
j = 1: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81]
j = 1: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 2: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 3: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 4: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Iterasi ke-7
j = 1: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 2: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
j = 3: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81] (Tidak ada pertukaran)
Array terurut: [3, 4, 7, 10, 12, 13, 16, 26, 75, 81]
```

Menganalisis Algoritma

```
For i ← 1 to n do
For k ← n downto 1+1 do
        If data[k] < data[k-1] then
                temp ← data[k]
                data[k] ← data[k − 1]
                data[k − 1] ← temp
        endif
endfor
endfor
```

Berdasarkan larik data dengan 6 buah elemen yang berisi angka-angka yang random (7,10,3,6,2,8)

Tuliskan hasil proses dari algoritma di atas pada setiap perulangan nilai i!

```
Data awal: [7, 10, 3, 6, 2, 8]
Iterasi ke-1
k = 5
8 < 2 FALSE
Data sementara: [7, 10, 3, 6, 2, 8]
k = 4
2 < 6 TRUE (swap)
Data sementara: [7, 10, 3, 2, 6, 8]
k = 3
2 < 3 TRUE (swap)
```

```
Data sementara: [7, 10, 2, 3, 6, 8]
k = 2
2 < 10 TRUE (swap)
Data sementara: [7, 2, 10, 3, 6, 8]
k = 1
2 < 7 TRUE (swap)
Data sementara: [2, 7, 10, 3, 6, 8]
Iterasi ke-2
k = 5
8 < 6 FALSE
Data sementara: [2, 7, 10, 3, 6, 8]
k = 4
6 < 3 FALSE
Data sementara: [2, 7, 10, 3, 6, 8]
k = 3
3 < 10 TRUE (swap)
Data sementara: [2, 7, 3, 10, 6, 8]
k = 2
3 < 7 TRUE (swap)
Data sementara: [2, 3, 7, 10, 6, 8]
Iterasi ke-3
k = 5
8 < 6 FALSE
Data sementara: [2, 3, 7, 10, 6, 8]
k = 4
6 < 10 TRUE (swap)
Data sementara: [2, 3, 7, 6, 10, 8]
k = 3
6 < 7 TRUE (swap)
Data sementara: [2, 3, 6, 7, 10, 8]
Iterasi ke-4
k = 5
8 < 10 TRUE (swap)
Data sementara: [2, 3, 6, 7, 8, 10]
k = 4
8 < 7 FALSE
Data sementara: [2, 3, 6, 7, 8, 10]
Iterasi ke-5
k = 5
10 < 8 FALSE
Data sementara: [2, 3, 6, 7, 8, 10]
Iterasi ke-6
Data akhir: [2, 3, 6, 7, 8, 10]
```

- Terdapat penugasan 4 orang ke 4 pekerjaan (job) dengan matriks biaya terlihat pada gambar di bawah. Setiap orang akan ditugasi mengerjakan satu pekerjaan. Persoalannya adalah bagaimana melakukan penugasan tersebut sehingga menghasilkan total biaya penugasan seminimal mungkin.

Job 1	Job 2	Job 3	Job 4	
9	2	7	8	Orang a
6	4	3	7	Orang b
5	8	1	4	Orang c
7	6	9	4	Orang d

a) Bila menggunakan strategi Brute Force tentukan berapa banyak percobaan yang harus dicoba

$$Complexity = O(n!) = O(4!) = O(24)$$

b) Bila menggunakan strategi greedy tentukan caranya dan tentukan hasilnya

Job 1	Job 2	Job 3	Job 4	
9	2	7	8	Orang a
6	4	3	7	Orang b
5	8	1	4	Orang c
7	6	9	4	Orang d

Untuk menyelesaikan permasalahan penugasan dengan menggunakan strategi greedy, kita dapat mengikuti langkah-langkah berikut:

1) Mengidentifikasi $e_{terkecil}$ dalam matriks biaya dan menempatkan orang pada pekerjaan dengan biaya terkecil

2) Menghapus baris dan kolom yang bersesuaian dengan penugasan tersebut
3) Ulangi Langkah 1-3 hingga semua orang telah ditugaskan

c -> Job 3 (biaya: 1)
a -> Job 2 (biaya: 2)
d -> Job 4 (biaya: 4)
b -> Job 1 (biaya: 6)
Total biaya penugasan: 13

- Terdapat penugasan 5 orang ke 5 pekerjaan (job) dengan matriks produktivitas terlihat pada gambar di bawah. Setiap orang hanya ditugasi satu pekerjaan. Persoalannya adalah bagaimana melakukan penugasan tersebut sehingga menghasilkan produktivitas **semaksimal mungkin**. Dengan menggunakan strategi Brute Force

	P1	P2	P3	P4	P5
Job1	8	7	4	9	5
Job2	4	2	4	5	7
Job3	9	5	8	7	8
Job4	4	3	7	6	7
Job5	6	7	4	6	5

a) Tentukan berapa banyak percobaan yang harus dicoba (20 point)

$$Complexity = O(n!) = O(5!) = O(120)$$

b) Tentukan nilai produktivitas maksimal yang didapat!

	P1	P2	P3	P4	P5
Job1	8	7	4	9	5
Job2	4	2	4	5	7
Job3	9	5	8	7	8
Job4	4	3	7	6	7
Job5	6	7	4	6	5

P4 -> Job1 (produktivitas: 9)
P5 -> Job2 (produktivitas: 7)
P1 -> Job3 (produktivitas: 9)
P3 -> Job4 (produktivitas: 7)
P2 -> Job5 (produktivitas: 7)
Produktivitas maksimum adalah: 39

- Sebuah kotak dapat diisi dengan fraksi obyek-obyek. Kapasitas kotak adalah 30 kg, sedangkan 6 obyek yang akan dimasukkan masing-masing mempunyai massa (satuan kg) 5,12,15,18, 20, 25.
  a) Tentukan komposisi obyek-obyek yang dimasukkan ke dalam kotak sehingga total nilai obyek di dalamnya maksimum!

$$\sum n_{berat} = \text{capacity}$$

Komposisi obyek

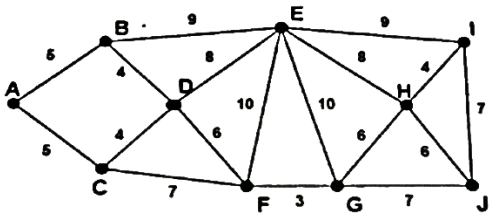
$\{5,25\}$

$\{12,18\}$

  b) Tuliskan algoritma greedy untuk penyelesaian kasus di atas!

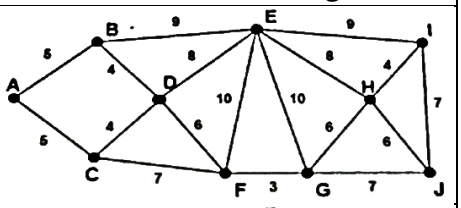
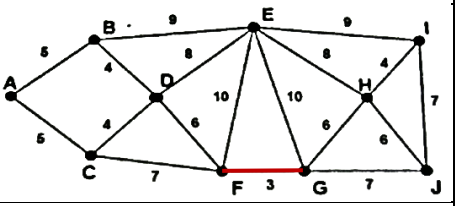
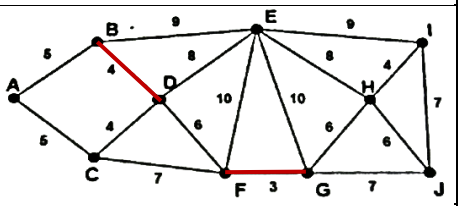
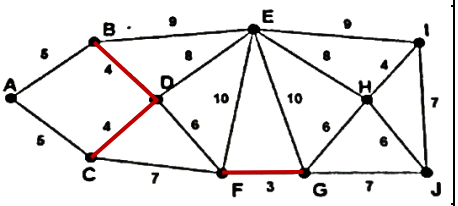
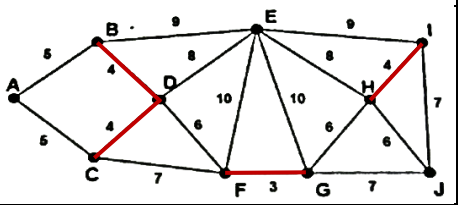
```
Procedure AssignJobs (input/output cost_matrix: 2D array of integers, input/output people: array of
strings, input/output jobs: array of strings)
{Mengalokasikan pekerjaan kepada orang dengan biaya minimum menggunakan pendekatan greedy.
Masukkan: Matriks biaya yang berisi biaya penugasan setiap pekerjaan ke setiap orang,
Daftar orang dan pekerjaan yang belum ditugaskan.
Keluaran: Daftar penugasan yang berisi pasangan orang dan pekerjaan serta biaya penugasannya.}

Deklarasi
i       Integer (indeks untuk baris dalam matriks biaya)
j       Integer (indeks untuk kolom dalam matriks biaya)
min_costInteger (biaya minimum dalam matriks biaya)
assignments     array of tuples  (daftar penugasan)

Algoritma
        While cost_matrix is not empty do
                {Menemukan biaya minimum dalam matriks biaya}
                min_cost ← minimum value in cost_matrix
                For i from 0 to length of cost_matrix do
                        If min_cost is in cost_matrix[i] then
                                j ← index of min_cost in cost_matrix[i]
                                Break the loop
                        Endif
                Endfor

                {Menambahkan penugasan ke dalam daftar}
                Add (people[i], jobs[j], min_cost) to assignments

                {Menghapus baris dan kolom yang bersesuaian dengan penugasan}
                Remove people[i] and jobs[j] from the list
                Remove i-th row and j-th column from cost_matrix
        Endwhile

        Return assignments
EndProcedure
```

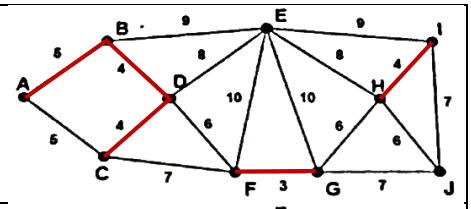
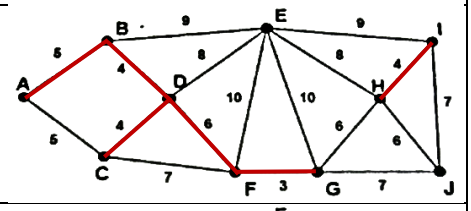
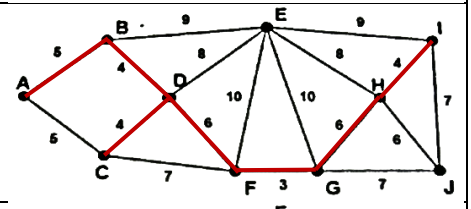
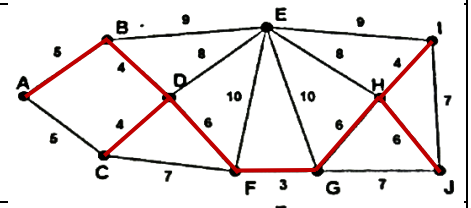
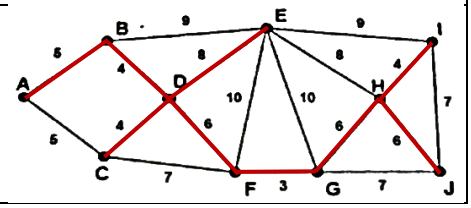
Algoritma Minimum Spanning Tree (MST)



a. Buatlah MST dengan *Algoritma Kurskal* dan tuliskan algoritmanya pada Graph di atas

```
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
```

Langkah	Sisi	Bobot	Hutan Merentang
0			
1	(F,G)	3	
2	(B,D)	4	
3	(D,C)	4	
4	(H,I)	4	

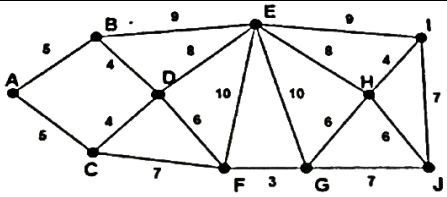
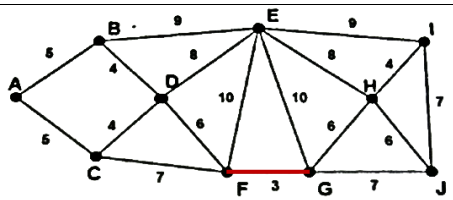
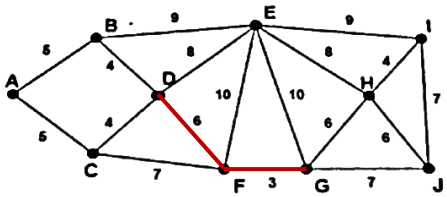
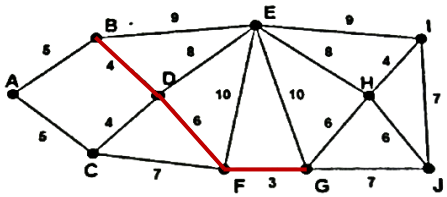
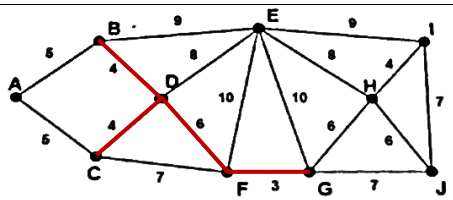
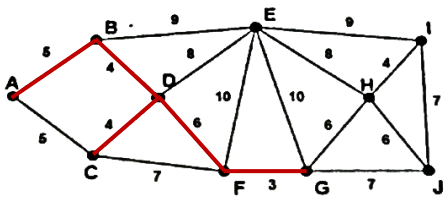
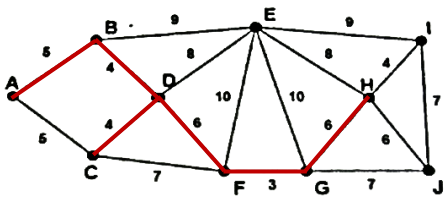
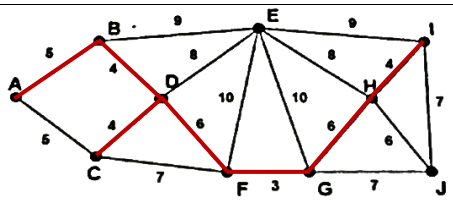
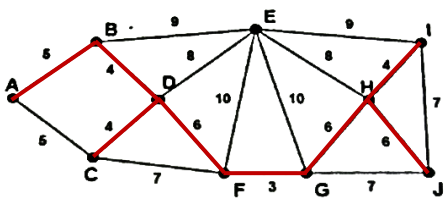
5	(A,B)	5	
6	(D,F)	6	
7	(G,H)	6	
8	(H,J)	6	
9	(D,E)	8	

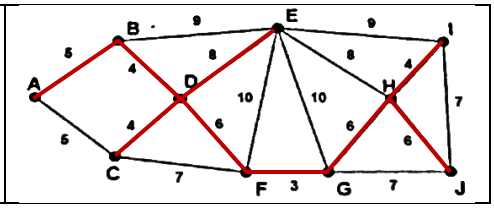
Sisi	(F,G)	(B,D)	(D,C)	(H,I)	(A,B)	(D,F)	(G,H)	(H,J)	(D,E)
Bobot	3	4	4	4	5	6	6	6	8

$Bobot = 3 + 4 + 4 + 4 + 5 + 6 + 6 + 6 + 8 = 46$

b. Buatlah MST dengan *Algoritma Prim's* dan tuliskan algoritmanya pada Graph di atas

```
Procedure Prim(input G: graf, output T: pohon)
{Memebentuk 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
Cari sisi (p,q) dari E yang berbobot terkecil
T ← {(p,q)}
for i ← 1 to n-1 do
    pilih ssi (u,v) dari E yang bobotnya terkecil namun bersisian dengan simpul T
    T ← TU{(u,v)}
Endfor
```

Langkah	Sisi	Bobot	Pohon Rentang
0			
1	(F,G)	3	
2	(D,F)	6	
3	(B,D)	4	
4	(D,C)	4	
5	(A,B)	5	
6	(G,H)	6	
7	(H,I)	4	
8	(H,J)	6	

9	(D,E)	8	

Sisi	(F,G)	(D,F)	(B,D)	(D,C)	(A,B)	(G,H)	(H,I)	(H,J)	(D,E)
Bobot	3	6	4	4	5	6	4	6	8

$Bobot = 3 + 6 + 4 + 4 + 5 + 6 + 4 + 6 + 8 = 46$

Jika diketahui nilai bobot(W), keuntungan(b), Kapasitas knapsack W-6:

- N=4.
- w1=3.5;b1=7
- w2=2.5;b2=5
- w3=5.5;b3=10
- w4=4.5;b4=8

a. Hitunglah keuntungan maksimum secara manual, menggunakan metode Brute force.

$$Complexity = O(2^n) = O(2^4) = 16$$

Kombinasi	**Bobot**	**Keuntungan**
Tidak ada item	0	0
Item 1	3.5	7
Item 2	2.5	5
Item 3	5.5	10
Item 4	4.5	8
Item 1, 2	6	12 (maksimum)
Item 1, 3	9 (Tidak Memenuhi)	17
Item 1, 4	8 (Tidak Memenuhi)	15
Item 2, 3	8 (Tidak Memenuhi)	15
Item 2, 4	7 (Tidak Memenuhi)	13
Item 3, 4	10 (Tidak Memenuhi)	18
Item 1, 2, 3	11.5 (Tidak Memenuhi)	22
Item 1, 2, 4	10.5 (Tidak Memenuhi)	20
Item 1, 3, 4	13.5 (Tidak Memenuhi)	25
Item 2, 3, 4	12.5 (Tidak Memenuhi)	23
Semua item	16 (Tidak Memenuhi)	30

b. Buatlah program soal di atas sehingga ditampilkan keuntungan maksimumnya.

```
from itertools import combinations

def knapsack_problem(weights, profits, W):
    comb = [combinations(range(len(weights)), r) for r in range(1, len(weights) + 1)]
```

```
    max_profit, best_comb = max(((sum(profits[i] for i in c), c) for combo in comb for c in combo if
sum(weights[i] for i in c) <= W), default=(0, None))
    best_items = [weights[i] for i in best_comb]
    return max_profit, best_comb, best_items

weights = [3.5, 2.5, 5.5, 4.5]
profits = [7, 5, 10, 8]
knapsack_capacity = 6

max_profit, best_comb, best_items = knapsack_problem(weights, profits, knapsack_capacity)

print(f"Keuntungan maksimum: {max_profit}")
print(f"Kombinasi barang terbaik: {best_comb}")
print(f"Berat barang terbaik: {best_items}")
```

Output:
Keuntungan maksimum: 12
Kombinasi barang terbaik: (0, 1)
Berat barang terbaik: [3.5, 2.5]

- Jika diketahui table A berisi elemen-elemen berikut: 5 12 3 9 1 20 7 2

Urutkan table di atas menggunakan:

a. Metode Divide and Conquer secara manual.

```
Urutkan array: [5, 12, 3, 9, 1, 20, 7, 2]
Pivot yang dipilih: 5
Elemen kurang dari atau sama dengan pivot: [3, 1, 2]
Elemen lebih besar dari pivot: [12, 9, 20, 7]
Menggabungkan [3, 1, 2], 5, dan [12, 9, 20, 7]
Urutkan array: [3, 1, 2]
Pivot yang dipilih: 3
Elemen kurang dari atau sama dengan pivot: [1, 2]
Elemen lebih besar dari pivot: []
Menggabungkan [1, 2], 3, dan []
Urutkan array: [1, 2]
Pivot yang dipilih: 1
Elemen kurang dari atau sama dengan pivot: []
Elemen lebih besar dari pivot: [2]
Menggabungkan [], 1, dan [2]
Urutkan array: []
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
Urutkan array: [2]
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
Array yang telah diurutkan: [1, 2]
Urutkan array: []
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
Array yang telah diurutkan: [1, 2, 3]
Urutkan array: [12, 9, 20, 7]
Pivot yang dipilih: 12
Elemen kurang dari atau sama dengan pivot: [9, 7]
Elemen lebih besar dari pivot: [20]
Menggabungkan [9, 7], 12, dan [20]
Urutkan array: [9, 7]
Pivot yang dipilih: 9
Elemen kurang dari atau sama dengan pivot: [7]
Elemen lebih besar dari pivot: []
Menggabungkan [7], 9, dan []
Urutkan array: [7]
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
Urutkan array: []
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
Array yang telah diurutkan: [7, 9]
Urutkan array: [20]
Array berisi 1 atau kurang elemen, kembalikan seperti adanya.
```

```
Array yang telah diurutkan: [7, 9, 12, 20]
Array yang telah diurutkan: [1, 2, 3, 5, 7, 9, 12, 20]
[1, 2, 3, 5, 7, 9, 12, 20]
```

b. Buatlah program sehingga ditampilkan elemen-elemen di atas secara urut (Ascending).

```
def quick_sort(arr):

    if len(arr) <= 1:

        return arr

    else:

        pivot = arr[0]

        less_than_pivot = [x for x in arr[1:] if x <= pivot]

        greater_than_pivot = [x for x in arr[1:] if x > pivot]

        return quick_sort(less_than_pivot) + [pivot] + quick_sort(greater_than_pivot)


arr = [5, 12, 3, 9, 1, 20, 7, 2]

print(quick_sort(arr))
```

Kompleksitas Waktu'

Sebuah algoritma tidak saja harus benar sesuai spesifikasi persoalan tetapi juga harus sangkil efisien

Algoritma yang bagus adalah algoritma yang sangkil efficient

Kesangkilan algoritma diukur dari waktu time yang diperlukan untuk menjalankan algoritma dan ruang space memori yang dibutuhkan oleh algoritma tersebut

Algoritma yang sangkil ialah algoritma yang meminimumkan kebutuhan waktu dan ruang memori

Kebutuhan waktu dan ruang memori suatu algoritma bergantung pada ukuran masukan n yang menyatakan ukuran data yang diproses oleh algoritma

Kesangkilan algoritma dapat digunakan untuk menilai algoritma yang bagus dari sejumlah algoritma penyelesaian persoalan

Sebab sebuah persoalan dapat memiliki banyak algoritma penyelesaian Contoh persoalan pengurutan sort ada puluhan algoritma pengurutan selection sort insertion sort bubble sort dll

Model Perhitungan Kebutuhan Waktu

Menghitung kebutuhan waktu algoritma dengan mengukur waktu eksekusi riil nya dalam satuan detik ketika program (yang merepresentasikan sebuah algoritma dijalankan oleh komputer bukanlah cara yang tepat

Alasan

Setiap komputer dengan arsitektur berbeda memiliki bahasa mesin yang

berbeda

waktu setiap operasi antara satu komputer dengan komputer lain tidak sama

Compiler bahasa pemrograman yang berbeda menghasilkan kode Bahasa mesin yang berbeda

waktu setiap operasi antara compiler dengan compiler lain tidak sama

Kompleksitas waktu, T(n), diukur dari jumlah tahapan komputasi yang dilakukan di dalam algoritma sebagai fungsi dari ukuran masukan n.

Kompleksitas ruang, S(n), diukur dari memori yang digunakan oleh struktur data

yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n.

Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan laju peningkatan waktu (ruang) yang diperlukan algoritma dengan meningkatnya ukuran masukan n.

Di dalam kuliah ini kita hanya membatasi bahasan kompleksitas waktu saja, karena dua alasan:

1) Materi struktur data diluar lingkup mata kuliah matematika diskrit
2) Saat ini memori komputer bukan persoalan yang kritis dibandingkan waktu

Pekerjaan utama di dalam kompleksitas waktu adalah menghitung (counting) jumlah

tahapan komputasi di dalam algoritma .

Jumlah tahapan komputasi dihitung dari berapa kali suatu operasi dilakukan sebagai fungsi ukuran masukan (n).

Di dalam sebuah algoritma terdapat banyak jenis operasi:

- Operasi baca/tulis (input a, print a)
- Operasi aritmetika (+, -, *, /) (a + b, M * N)
- Operasi pengisian nilai (assignment) (a <- 10)
- Operasi perbandingan (a < b, k >= 10)
- Operasi pengaksesan elemen larik, pemanggilan prosedur/fungsi, dll

Untuk menyederhanakan perhitungan, kita tidak menghitung semua jenis operasi, tetapi kita hanya menghitung jumlah operasi khas (tipikal) yang mendasari suatu algoritma.

Kategorisasi Kompleksitas Waktu

- O(1) - Konstan: tidak bergantung pada ukuran masukan.

- O(log n) - Logaritmik: seperti algoritma pencarian biner.

- O(n) - Linier: seperti pencarian linier, memproses setiap elemen masukan.

- O(n log n) - Linier logaritmik: divide-and-conquer, seperti merge sort.

- O(n^2) - Kuadratik: seperti selection sort, insertion sort.

- O(n^3) - Kubik: seperti perkalian matriks.

- O(2^n) - Eksponensial: seperti mencari solusi dengan mencoba semua kemungkinan.

- O(n!) - Faktorial: seperti algoritma persoalan pedagang keliling.

Made with by Rendie

Baik, berikut penjelasan yang lebih rinci tentang analisis kompleksitas waktu algoritma:

1. Memahami Konsep Dasar

- Kompleksitas waktu adalah analisis berapa lama waktu yang dibutuhkan oleh sebuah algoritma untuk menyelesaikan tugasnya sebagai fungsi dari ukuran masukan.

- Notasi asimptotik digunakan untuk menggambarkan kompleksitas waktu untuk ukuran masukan yang besar (mendekati tak hingga).

2. Notasi Big-O (O-besar)

- Notasi O-besar memberikan batas atas (upper bound) dari kompleksitas waktu algoritma.

- Jika T(n) = O(f(n)), artinya ada konstanta C dan nilai n0 sehingga T(n) ≤ C.f(n) untuk setiap n ≥ n0.

- Contoh:

- T(n) = 2n^2 + 6n + 1.

Karena 2n^2 + 6n + 1 ≤ 9n^2 untuk n ≥ 1 (C = 9, f(n) = n^2, n0 = 1), maka T(n) = O(n^2).

- Mencari nilai maksimum dari n elemen memiliki T(n) = n - 1 = O(n).

3. Notasi Big-Omega (Ω-besar)

- Notasi Ω-besar memberikan batas bawah (lower bound) dari kompleksitas waktu algoritma.

- Jika T(n) = Ω(g(n)), artinya ada konstanta C dan nilai n0 sehingga T(n) ≥ C.g(n) untuk setiap n ≥ n0.

- Contoh: T(n) = 2n^2 + 6n + 1 = Ω(n^2), karena 2n^2 + 6n + 1 ≥ 2n^2 untuk n ≥ 1.

Notasi O-Besar (Big-O)

- Notasi "O" disebut notasi "O-Besar" (*Big-O*) yang merupakan notasi kompleksitas waktu asimptotik.

- **DEFINISI 1.** $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n))$, yang artinya *$T(n)$ berorde paling besar $f(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga

  $T(n) \le C f(n)$

  untuk $n \ge n_0$.

- $f(n)$ adalah batas lebih atas (*upper bound*) dari $T(n)$ untuk n yang besar.

DEFINISI. $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n))$" yang artinya *$T(n)$ berorde paling besar $f(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga

$T(n) \le C(f(n))$

untuk $n \ge n_0$.

- **Catatan**: Ada tak-berhingga nilai C dan n_0 yang memenuhi $T(n) \le C f(n)$, kita cukup menunjukkan satu pasang (C, n_0) yang memenuhi definisi sehingga $T(n) = O(f(n))$

Contoh 7. Tunjukkan bahwa $2n^2 + 6n + 1 = O(n^2)$. (tanda '=' dibaca 'adalah')

Penyelesaian:

$2n^2 + 6n + 1 = O(n^2)$ karena

$2n^2 + 6n + 1 \le 2n^2 + 6n^2 + n^2 = 9n^2$ untuk semua $n \ge 1$ ($C = 9$, $f(n) = n^2$, $n_0 = 1$).

atau karena

$2n^2 + 6n + 1 \le n^2 + n^2 + n^2 = 3n^2$ untuk semua $n \ge 7$ ($C = 3$, $f(n) = n^2$, $n_0 = 7$).

DEFINISI. $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n))$" yang artinya *$T(n)$ berorde paling besar $f(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga

$T(n) \le C(f(n))$

untuk $n \ge n_0$.

Contoh 8. Tunjukkan bahwa $3n + 2 = O(n)$.

Penyelesaian:

$3n + 2 = O(n)$

karena

$3n + 2 \le 3n + 2n = 5n$ untuk semua $n \ge 1$

($C = 5$, $f(n) = n$, dan $n_0 = 1$).

DEFINISI. $T(n) = O(f(n))$ (dibaca "$T(n)$ adalah $O(f(n))$" yang artinya *$T(n)$ berorde paling besar $f(n)$*) bila terdapat konstanta C dan n_0 sedemikian sehingga

$T(n) \le C(f(n))$

untuk $n \ge n_0$.

Contoh-contoh Lain

1. Tunjukkan bahwa $5 = O(1)$.

Jawaban:

$5 = O(1)$ karena $5 \le 6 \cdot 1$ untuk $n \ge 1$ ($C = 6$, $f(n) = 1$, dan $n_0 = 1$)

Kita juga dapat memperlihatkan bahwa

$5 = O(1)$ karena $5 \le 10 \cdot 1$ untuk $n \ge 1$ ($C = 10$, $f(n) = 1$, dan $n_0 = 1$)

2. Tunjukkan bahwa kompleksitas waktu algoritma pengurutan seleksi (*selection sort*) adalah $T(n) = \frac{n(n-1)}{2} = O(n^2)$.

Jawaban:

$\frac{n(n-1)}{2} = O(n^2)$

karena

$\frac{n(n-1)}{2} \le \frac{1}{2}n^2 + \frac{1}{2}n^2 = n^2$

untuk $n \ge 1$

($C = 1$, $f(n) = n^2$, dan $n_0 = 1$).

3. Tunjukkan $6 \cdot 2^n + 2n^2 = O(2^n)$

Jawaban:

$6 \cdot 2^n + 2n^2 = O(2^n)$

karena

$6 \cdot 2^n + 2n^2 \le 6 \cdot 2^n + 2 \cdot 2^n = 8 \cdot 2^n$

untuk semua $n \ge 4$ ($C = 8$, $f(n) = 2^n$, dan $n_0 = 4$).

4. Tunjukkan $1 + 2 + \ldots + n = O(n^2)$

Jawaban:

Cara 1: $1 + 2 + \ldots + n \le n + n + \ldots + n = n^2$ untuk $n \ge 1$

Cara 2: $1 + 2 + \ldots + n = \frac{1}{2}n(n+1) \le \frac{1}{2}n^2 + \frac{1}{2}n^2 = n^2$ untuk $n \ge 1$

5. Tunjukkan $n! = O(n^n)$

Jawaban:

$n! = 1 \cdot 2 \cdot \ldots \cdot n \le n \cdot n \cdot \ldots \cdot n = n^n$ untuk $n \ge 1$

Teorema 1: Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ adalah polinom derajat $\le m$ maka $T(n) = O(n^m)$.

- Jadi, untuk menentukan notasi *Big-Oh*, cukup melihat suku (*term*) yang mempunyai pangkat terbesar di dalam $T(n)$.

- **Contoh 8**:

  $T(n) = 5 = 5n^0 = O(n^0) = O(1)$

  $T(n) = 2n + 3 = O(n)$

  $T(n) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$

  $T(n) = 3n^3 + 2n^2 + 10 = O(n^3)$

Tunjukkan bahwa $T(n) = 5n^2 = O(n^3)$, tetapi $T(n) = n^3 \neq O(n^2)$.

Jawaban:

- $5n^2 = O(n^3)$ karena $5n^2 \le n^3$ untuk semua $n \ge 5$.

- Tetapi, $T(n) = n^3 \neq O(n^2)$ karena tidak ada konstanta C dan n_0 sedemikian sehingga $n^3 \le Cn^2 \Leftrightarrow n \le C$ untuk semua n_0 karena n dapat berupa sembarang bilangan yang besar.

- Menuliskan

  $O(2n)$ tidak standard, seharusnya $O(n)$

  $O(n-1)$ tidak standard, seharusnya $O(n)$

  $O(\frac{n^2}{2})$ tidak standard, seharusnya $O(n^2)$

  $O((n-1)!)$ tidak standard, seharusnya $O(n!)$

- Ingat, di dalam notasi Big-Oh tidak ada koefisien atau suku-suku lainnya, hanya berisi fungsi-fungsi standard seperti $1, n^2, n^3, \ldots, \log n, n \log n, 2^n, n!$, dan sebagainya

Contoh 11: Tentukan notasi O-besar untuk $T(n) = (n+1)\log(n^2+1) + 3n^2$.

Jawaban:

Cara 1:
- $n + 1 = O(n)$
- $\log(n^2+1) \le \log(2n^2) = \log(2) + \log(n^2)$
  $= \log(2) + 2\log(n)$
  $\le \log(n) + 2\log(n) = 3\log(n)$ untuk $n \ge 2$
  $= O(\log n)$
- $(n+1)\log(n^2+1) = O(n)\,O(\log n) = O(n \log n)$
- $3n^2 = O(n^2)$
- $(n+1)\log(n^2+1) + 3n^2 = O(n \log n) + O(n^2) = O(\max(n \log n, n^2)) = O(n^2)$

Cara 2: suku yang dominan di dalam $(n+1)\log(n^2+1) + 3n^2$ untuk n yang besar adalah $3n^2$, sehingga $(n+1)\log(n^2+1) + 3n^2 = O(n^2)$

6. Tunjukkan $\log n! = O(n \log n)$

Jawaban:

Dari soal 5 sudah diperoleh bahwa $n! \le n^n$ untuk $n \ge 1$ maka
$\log n! \le \log n^n = n \log n$ untuk $n \ge 1$maka
sehingga $\log n! = O(n \log n)$

7. Tunjukkan $8n^2 = O(n^3)$

Jawaban:

$8n^2 = O(n^3)$ karena $8n^2 \le n^3$ untuk $n \ge 8$

- Teorema 1 tersebut digeneralisasi untuk suku-suku dominan lainnya:
  1. Eksponensial mendominasi sembarang perpangkatan (yaitu, $y^n > n^p$, $y > 1$)
  2. Perpangkatan mendominasi $\ln(n)$ (yaitu $n^p > \ln n$)
  3. Semua logaritma tumbuh pada laju yang sama (yaitu $a \log(n) = b \log(n)$)
  4. $n \log n$ tumbuh lebih cepat daripada n tetapi lebih lambat daripada n^2

Contoh 9: $T(n) = 2^n + 2n^2 = O(2^n)$.
$T(n) = 2n \log(n) + 3n = O(n \log n)$
$T(n) = \log n^3 = 3 \log(n) = O(\log n)$
$T(n) = 2n \log n + 3n^2 = O(n^2)$

- Definisi: $T(n) = O(f(n))$ jika terdapat C dan n_0 sedemikian sehingga $T(n) \le C f(n)$ untuk $n \ge n_0$
  → tidak menyiratkan seberapa atas fungsi f itu.

- Jadi, menyatakan bahwa
  $T(n) = 2n^2 = O(n^2)$ → benar
  $T(n) = 2n^2 = O(n^3)$ → juga benar, karena $2n^2 \le 2n^3$ untuk $n \ge 1$
  $T(n) = 2n^2 = O(n^4)$ → juga benar, karena $2n^2 \le 2n^4$ untuk $n \ge 1$
- Namun, untuk alasan praktikal kita memilih fungsi yang sekecil mungkin agar O(f(n)) memiliki makna
- Jadi, kita menulis $2n^2 = O(n^2)$, bukan $O(n^3)$ atau $O(n^4)$

TEOREMA 2. Misalkan $T_1(n) = O(f(n))$ dan $T_2(n) = O(g(n))$, maka
(a) $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$
(b) $T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$
(c) $O(cf(n)) = O(f(n))$, c adalah konstanta
(d) $f(n) = O(f(n))$

Contoh 9. Misalkan $T_1(n) = O(n)$ dan $T_2(n) = O(n^2)$, maka
(a) $T_1(n) + T_2(n) = O(\max(n, n^2)) = O(n^2)$
(b) $T_1(n)T_2(n) = O(nn^2) = O(n^3)$

Contoh 10. $O(5n^2) = O(n^2)$
$n^2 = O(n^2)$

Contoh 3:

```
for i ← 1 to n do
  for j ← 1 to i do
      a ← a + 1      O(1)
      b ← b – 2      O(1)
  endfor
endfor
```

Kompleksitas untuk $a \leftarrow a + 1$ $= O(1)$
Kompleksitas untuk $b \leftarrow b - 2$ $= O(1)$
Kompleksitas total keduanya $= O(1) + O(1) = O(1)$
Jumlah pengulangan seluruhnya $= 1 + 2 + \ldots + n = n(n+1)/2$
Kompleksitas seluruhnya $= n(n+1)/2\ O(1) = O(n(n+1)/2)$
$= O(n^2/2 + n/2)$
$= O(n^2)$

1. Apa yang dimaksud dengan algoritma brute force?
   a) Algoritma yang menggunakan kecerdasan buatan untuk menyelesaikan masalah
   b) Algoritma yang mencari solusi optimal secara langsung
   c) Algoritma yang mencoba semua kemungkinan solusi secara sistematis
   d) Algoritma yang hanya dapat digunakan untuk masalah kecil
2. Ketika menggunakan algoritma brute force, apa yang terjadi jika jumlah kemungkinan solusi sangat besar?
   a) Algoritma akan memberikan solusi optimal
   b) Algoritma akan berhenti dan tidak memberikan solusi
   c) Algoritma akan memakan waktu yang sangat lama untuk mencari solusi
   d) Algoritma akan memberikan solusi yang salah
3. Apa kelemahan utama dari algoritma brute force?
   a) Memerlukan perangkat keras yang kuat
   b) Memerlukan pemrogram yang sangat ahli
   c) Memerlukan waktu yang sangat lama untuk menyelesaikan masalah dengan ukuran yang besar
   d) Tidak dapat menyelesaikan masalah apapun
4. Bagaimana cara mengidentifikasi apakah algoritma brute force merupakan solusi yang tepat untuk suatu masalah?
   a) Dengan mencoba algoritma brute force terlebih dahulu
   b) Dengan menganalisis kompleksitas waktu algoritma brute force
   c) Dengan memilih algoritma brute force jika tidak ada solusi yang lain
   d) Dengan menggunakan algoritma brute force untuk setiap masalah
5. Kapan algoritma brute force biasanya digunakan?
   a) Ketika solusi optimal diperlukan dengan cepat
   b) Ketika masalah memiliki jumlah kemungkinan solusi yang terbatas
   c) Ketika tidak ada algoritma yang lain yang dapat digunakan
   d) Ketika memerlukan solusi untuk masalah dengan jumlah kemungkinan solusi yang kecil
6. Apa perbedaan utama antara algoritma brute force dan greedy?
   a) Algoritma brute force mencari solusi optimal, sedangkan greedy mencari solusi yang cukup baik
   b) Algoritma greedy mencoba semua kemungkinan solusi, sedangkan brute force hanya mencoba solusi yang terdekat
   c) Algoritma brute force lebih cepat daripada greedy
   d) Tidak ada perbedaan antara keduanya
7. Bagaimana kompleksitas waktu dari algoritma brute force dan greedy?
   a) Brute force memiliki kompleksitas waktu yang lebih rendah daripada greedy
   b) Greedy memiliki kompleksitas waktu yang lebih rendah daripada brute force
   c) Keduanya memiliki kompleksitas waktu yang sama
   d) Tergantung pada masalah yang diselesaikan
8. Ketika memilih solusi, algoritma brute force akan:
   a) Memilih solusi yang optimal pada setiap langkah
   b) Memilih solusi yang terlihat paling baik pada setiap langkah
   c) Memilih solusi yang paling sederhana pada setiap langkah
   d) Memilih solusi secara acak pada setiap langkah
9. Apa yang dilakukan oleh algoritma greedy ketika memilih solusi?
   a) Menimbang semua kemungkinan solusi
   b) Memilih solusi yang paling optimal secara global pada setiap langkah
   c) Memilih solusi yang paling baik secara lokal pada setiap langkah
   d) Memilih solusi yang paling kompleks pada setiap langkah
10. Algoritma brute force lebih cocok digunakan ketika:
    a) Masalah memiliki jumlah kemungkinan solusi yang sangat besar
    b) Masalah memiliki jumlah kemungkinan solusi yang terbatas
    c) Masalah hanya memiliki satu solusi yang mungkin
    d) Masalah memiliki solusi yang tidak terdefinisi
11. Dalam hal kecepatan, algoritma greedy biasanya lebih cepat daripada brute force karena:
    a) Greedy hanya mempertimbangkan solusi yang terbaik pada setiap langkah
    b) Greedy tidak perlu melakukan backtracking
    c) Greedy tidak perlu mempertimbangkan semua kemungkinan solusi
    d) Semua jawaban di atas benar
12. Algoritma greedy cenderung menghasilkan solusi yang optimal global?
    a) Ya, karena selalu memilih solusi terbaik pada setiap langkah
    b) Tidak, karena hanya mempertimbangkan keuntungan lokal pada setiap langkah
    c) Tergantung pada kompleksitas masalah
    d) Hanya dalam beberapa kasus khusus
13. Dalam hal kompleksitas ruang (space complexity), algoritma brute force biasanya:
    a) Memiliki kompleksitas ruang yang lebih rendah daripada greedy
    b) Memiliki kompleksitas ruang yang lebih tinggi daripada greedy
    c) Keduanya memiliki kompleksitas ruang yang sama
    d) Tergantung pada implementasi masing-masing algoritma
14. Kapan algoritma greedy biasanya digunakan?
    a) Ketika solusi optimal diperlukan
    b) Ketika masalah memiliki jumlah kemungkinan solusi yang besar
    c) Ketika solusi yang cukup baik sudah memadai
    d) Ketika masalah memerlukan semua kemungkinan solusi untuk dijelajahi
15. Algoritma brute force sering digunakan dalam konteks:
    a) Kriptografi
    b) Perutean (Routing)
    c) Optimasi fungsi matematika
    d) Semua jawaban di atas benar
16. Apa yang menjadi fokus utama dari algoritma Divide and Conquer?
    a) Mengurangi masalah menjadi ukuran yang lebih kecil pada setiap iterasi
    b) Memecah masalah menjadi submasalah yang lebih kecil, menyelesaikan setiap submasalah, dan menggabungkan solusinya
    c) Mengurangi kompleksitas waktu dengan mengurangi jumlah langkah yang diperlukan untuk menyelesaikan masalah
    d) Tidak ada perbedaan antara algoritma Divide and Conquer dengan Decrease and Conquer

17. Apa prinsip dasar dari algoritma Decrease and Conquer?
    a) Membagi masalah menjadi submasalah yang lebih kecil, menyelesaikan setiap submasalah, dan menggabungkan solusinya
    b) Mengurangi masalah menjadi ukuran yang lebih kecil pada setiap iterasi
    c) Mencari pola di dalam masalah dan mengurangi ukurannya secara berurutan
    d) Tidak ada perbedaan antara keduanya, keduanya mengacu pada prinsip yang sama
18. Pada tahap mana algoritma Divide and Conquer membagi masalah menjadi submasalah yang lebih kecil?
    a) Tahap "divide"
    b) Tahap "conquer"
    c) Tahap "combine"
    d) Tahap "merge"
19. Pada tahap mana algoritma Decrease and Conquer mengurangi masalah menjadi ukuran yang lebih kecil?
    a) Tahap "divide"
    b) Tahap "conquer"
    c) Tahap "reduce"
    d) Tahap "merge"
20. Algoritma Decrease and Conquer cenderung menyelesaikan masalah dengan cara
    a) Memecah masalah menjadi submasalah yang lebih kecil
    b) Mengurangi masalah menjadi ukuran yang lebih kecil pada setiap iterasi
    c) Menggabungkan solusi dari submasalah yang lebih kecil
    d) Menyelesaikan masalah secara langsung tanpa membaginya
21. Algoritma Divide and Conquer lebih cocok digunakan untuk masalah yang memiliki:
    a) Struktur hierarkis dan dapat didekomposisi ke dalam submasalah yang lebih kecil
    b) Masalah yang dapat dipecah menjadi bagian-bagian yang berbeda
    c) Masalah yang memerlukan pengurangan ukuran pada setiap iterasi
    d) Masalah yang hanya memiliki satu solusi yang mungkin
22. Proses reduksi dalam algoritma Decrease and Conquer dilakukan pada tahap:
    a) "divide"
    b) "conquer"
    c) "reduce"
    d) "combine"
23. Dalam algoritma Decrease and Conquer, proses "decrease" mengacu pada:
    a) Membagi masalah menjadi submasalah yang lebih kecil
    b) Mengurangi masalah menjadi ukuran yang lebih kecil pada setiap iterasi
    c) Menyelesaikan setiap submasalah
    d) Menggabungkan solusi dari submasalah yang lebih kecil

24. Algoritma Divide and Conquer sering memiliki kompleksitas waktu yang:
    a) Lebih rendah daripada Decrease and Conquer
    b) Lebih tinggi daripada Decrease and Conquer
    c) Sama dengan Decrease and Conquer
    d) Bergantung pada masalah yang dihadapi

25. Algoritma Decrease and Conquer cenderung lebih efisien daripada Divide and Conquer dalam kasus-kasus di mana:
    a) Masalah dapat dibagi menjadi bagian-bagian yang sama besar
    b) Masalah memiliki struktur hierarkis yang kompleks
    c) Masalah memiliki pola yang jelas dan dapat ditempatkan dalam Langkah langkah diskrit
    d) Tidak ada perbedaan efisiensi antara keduanya