

ARTIKEL INFORMATIF
DEFINISI ALGORITMA DAN FUNGSI ALGORITMA DALAM PEMROGRAMAN
BESERTA DENGAN
JENIS-JENIS ALGORITMA BESERTA DAN CONTOHNYA



Yadit Firmansyah_D0424306

Disusun untuk memenuhi tugas matakuliah
algoritma dan struktur data

UNIVERSITAS SULAWESI BARAT

04/okt/2024

Abstrak

Algoritma merupakan komponen fundamental dalam pemrograman yang mendefinisikan langkah-langkah untuk menyelesaikan suatu masalah. Artikel ini membahas definisi algoritma, fungsi algoritma dalam konteks pemrograman, serta menjelaskan berbagai jenis algoritma yang umum digunakan seperti algoritma rekursif, pengurutan, pencarian, greedy, backtracking, dan randomized. Setiap jenis algoritma akan disertai dengan contoh implementasi untuk memberikan gambaran yang jelas mengenai penerapan algoritma dalam praktik.

Kata Kunci

Algoritma Rekursif, Algoritma Pengurutan, Algoritma Pencarian, Algoritma Greedy, Algoritma Backtracking, Algoritma Randomized.

Pendahuluan

Algoritma adalah serangkaian langkah sistematis yang diikuti untuk mencapai tujuan tertentu dalam pemrograman. Dalam dunia teknologi informasi, algoritma memainkan peran penting dalam menyelesaikan berbagai masalah secara efisien. Setiap program yang ditulis memiliki algoritma yang mendasarinya. Pengetahuan tentang berbagai jenis algoritma dan bagaimana cara kerjanya sangat diperlukan bagi programmer untuk menciptakan solusi yang optimal.

Metode

Metode yang digunakan dalam artikel ini adalah studi literatur, di mana penulis mengumpulkan informasi dari berbagai sumber, termasuk buku, artikel, dan sumber daring terkait algoritma dan penerapannya dalam pemrograman. Contoh kode ditulis menggunakan bahasa pemrograman Python untuk ilustrasi yang lebih jelas.

Hasil dan Pembahasan

1. Definisi Algoritma

Algoritma dapat didefinisikan sebagai serangkaian langkah sistematis yang harus diikuti untuk mencapai tujuan tertentu. Dalam konteks pemrograman, algoritma dapat berupa urutan instruksi yang ditulis dalam suatu bahasa pemrograman untuk menyelesaikan suatu masalah tertentu. Karakteristik utama dari algoritma adalah sebagai berikut:

- **Definiteness:** Langkah-langkah harus jelas dan pasti.
- **Input dan Output:** Algoritma harus dapat menerima input dan menghasilkan output.
- **Finiteness:** Algoritma harus berhenti setelah sejumlah langkah tertentu.
- **Efisiensi:** Algoritma yang baik harus meminimalkan waktu dan sumber daya

2. Fungsi algoritma dalam pemrograman

Algoritma memiliki beberapa fungsi penting dalam pemrograman, antara lain:

- **Memecahkan Masalah:** Algoritma membantu programmer untuk menemukan solusi untuk masalah yang dihadapi.
- **Meningkatkan Kinerja Program:** Algoritma yang efisien dapat meningkatkan kecepatan dan responsivitas program.
- **Menyederhanakan Proses:** Dengan membagi masalah menjadi langkah-langkah yang lebih kecil, algoritma dapat membuat pemecahan masalah lebih mudah dipahami.

3. Jenis-Jenis Algoritma dan Contohnya

3.1 Algoritma Rekursif (Recursive Algorithm)

Rekursif adalah fungsi yang memanggil dirinya sendiri secara langsung ataupun tidak. Tujuan rekursif adalah untuk melakukan pengulangan, atau looping seperti for dan while, namun dengan cara yang berbeda. Berikut ini contoh implementasi algoritma rekursif dalam Bahasa pemrograman Python:

```
# Algoritma Rekursif dalam Python
def rek(angka):
    if angka > 0:
        print(angka)
        angka = angka - 1
        rek(angka) # Panggilan rekursif
    else:
        print(angka)

# Meminta input dari user
n = int(input("Input Number: "))

# Memanggil fungsi rekursif dengan input n
rek(n)
```

Gambar 1.1 input Algoritma Rekursif

```
Input Number: 5
5
4
3
2
1
0
```

Gambar 1.2 output Algoritma Rekursif

Konsep Dasar Fungsi Rekursif

Pada dasarnya, sebuah fungsi rekursif terdiri dari dua komponen utama:

- Base Case (Kasus Dasar): Ini adalah kondisi di mana fungsi tidak akan memanggil dirinya sendiri lagi dan langsung memberikan hasil. Base case diperlukan untuk menghentikan rekursi dan mencegah terjadinya loop tak berujung.
- Recursive Case (Kasus Rekursif): Bagian ini adalah di mana fungsi memanggil dirinya sendiri dengan versi yang lebih sederhana atau lebih kecil dari masalah aslinya.

Contoh klasik dari fungsi rekursif adalah perhitungan faktorial dari suatu bilangan n . Faktorial dari n (ditulis sebagai $n!$) adalah produk dari semua bilangan bulat positif dari 1 hingga n . Ini dapat didefinisikan secara rekursif sebagai:

$n! = n \times (n-1)!$ dengan $0! = 1$ sebagai kasus dasar.

Berikut adalah implementasi fungsi rekursif untuk menghitung faktorial dalam Python:

```
def faktorial(n):  
    if n == 0:  
        return 1 # Base case  
    else:  
        return n * faktorial(n - 1) # Recursive case
```

Cara Kerja Fungsi Rekursif

Saat fungsi faktorial dipanggil dengan suatu nilai, misalnya $\text{faktorial}(5)$, fungsi ini akan terus memanggil dirinya sendiri dengan nilai n yang lebih kecil hingga mencapai base case ($\text{faktorial}(0)$). Setelah mencapai base case, nilai dikembalikan dan setiap panggilan fungsi sebelumnya akan dihitung hingga kembali ke panggilan fungsi awal.

Ilustrasi langkah-langkah untuk $\text{faktorial}(5)$:

$\text{faktorial}(5)$ memanggil $\text{faktorial}(4)$

$\text{faktorial}(4)$ memanggil $\text{faktorial}(3)$

$\text{faktorial}(3)$ memanggil $\text{faktorial}(2)$

$\text{faktorial}(2)$ memanggil $\text{faktorial}(1)$

$\text{faktorial}(1)$ memanggil $\text{faktorial}(0)$ yang mengembalikan 1

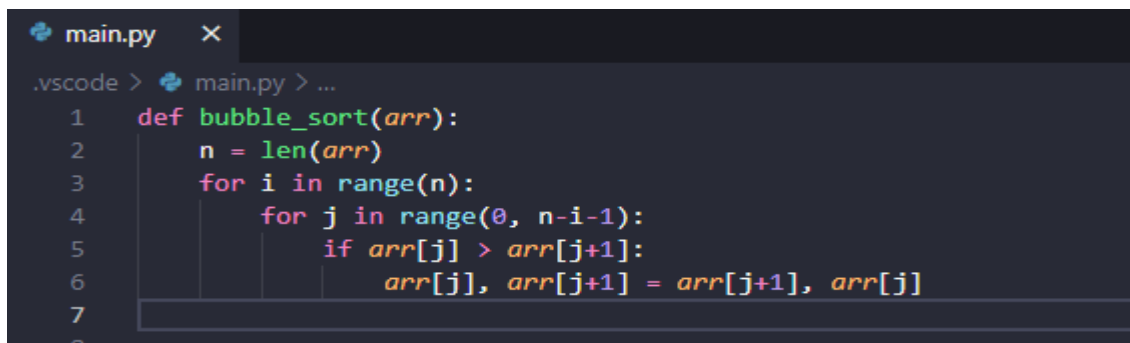
Hasilnya diurutkan kembali: $1 * 1 * 2 * 3 * 4 * 5 = 120$

3.2 Algoritma Sorting

Algoritma sorting (pengurutan) adalah metode untuk menyusun elemen-elemen dalam suatu daftar atau array berdasarkan urutan tertentu, baik secara ascending (naik) maupun descending (turun). Sorting sangat penting dalam berbagai aplikasi pemrograman karena memudahkan pencarian, pengurutan data, dan analisis yang lebih cepat.

Contoh Algoritma Sorting

1. **Bubble Sort:** Algoritma ini bekerja dengan membandingkan dua elemen berurutan dan menukar posisinya jika urutannya salah. Proses ini dilakukan berulang kali hingga tidak ada lagi elemen yang perlu ditukar.



```
main.py x
.vscode > main.py > ...
1  def bubble_sort(arr):
2      n = len(arr)
3      for i in range(n):
4          for j in range(0, n-i-1):
5              if arr[j] > arr[j+1]:
6                  arr[j], arr[j+1] = arr[j+1], arr[j]
7
```

Gambar 1.3 contoh bubble sort

Pada *Bubble Sort*, efisiensi rendah untuk jumlah data besar karena memerlukan banyak perbandingan dan pertukaran.

2. **Merge Sort** Merge sort adalah algoritma berbasis rekursi yang membagi array menjadi dua bagian, mengurutkannya secara terpisah, lalu menggabungkan kembali dua bagian yang sudah terurut.

Cara Kerja Merge Sort:

Divide: Membagi array atau daftar menjadi dua bagian hingga tidak dapat lagi dibagi (sampai mencapai satu elemen).

Conquer: Mengurutkan dua sub-array tersebut secara rekursif.

Combine: Menggabungkan dua sub-array yang sudah terurut menjadi satu array yang terurut.

Langkah-langkah Merge Sort:

Jika panjang array lebih dari satu, bagi array tersebut menjadi dua bagian.

Rekursif lakukan *merge sort* pada kedua bagian hingga masing-masing array memiliki satu elemen.

Setelah bagian terkecil diurutkan, gabungkan array kecil tersebut menjadi satu array yang lebih besar yang sudah terurut.

Contoh marge sort dalam python:

```
main.py X
.vscode > main.py > ...
1 def merge_sort(arr):
2     if len(arr) > 1:
3         mid = len(arr)//2
4         left_half = arr[:mid]
5         right_half = arr[mid:]
6
7         merge_sort(left_half)
8         merge_sort(right_half)
9
10        i = j = k = 0
11
12        while i < len(left_half) and j < len(right_half):
13            if left_half[i] < right_half[j]:
14                arr[k] = left_half[i]
15                i += 1
16            else:
17                arr[k] = right_half[j]
18                j += 1
19            k += 1
20
21        while i < len(left_half):
22            arr[k] = left_half[i]
23            i += 1
24            k += 1
25
26        while j < len(right_half):
27            arr[k] = right_half[j]
28            j += 1
29            k += 1
30
```

Gambar 1.4 marge short

3.3. algoritma searching

Algoritma searching berfungsi untuk mengambil informasi spesifik dari kumpulan data. , fungsi searching sering digunakan untuk mengidentifikasi dan menemukan data dari suatu kumpulan. Cara kerjanya tergantung pada struktur data yang digunakan serta algoritma yang dipilih. Efisiensi algoritma searching diukur dari seberapa cepat dan efisien ia dapat menemukan data, biasanya dinyatakan dalam notasi kompleksitas waktu $O(n)$.

Contoh Algoritma Searching

1. Linear Search

Algoritma ini bekerja dengan memeriksa setiap elemen dalam daftar satu per satu hingga elemen yang dicari ditemukan atau seluruh daftar diperiksa. Algoritma ini tidak efisien untuk dataset yang besar karena memerlukan waktu linear, yaitu $O(n)$.

Contohnya:

```
main.py X
.vscode > main.py > ...
1 def linear_search(arr, x):
2     for i in range(len(arr)):
3         if arr[i] == x:
4             return i
5     return -1
6
```

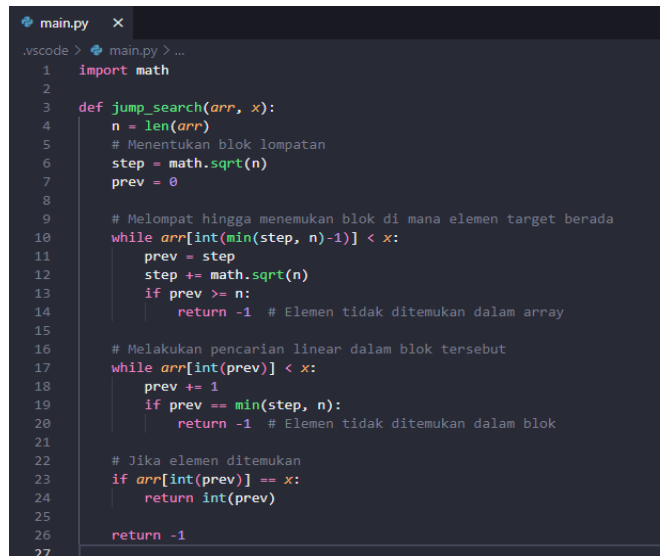
2. Binary Search

Algoritma ini bekerja lebih efisien dibanding linear search tetapi membutuhkan data yang sudah terurut. Ia mulai dari tengah-tengah daftar dan membagi ruang pencarian setiap kali hingga menemukan elemen yang dicari. Waktu yang dibutuhkan adalah $O(\log n)$, membuatnya lebih cepat untuk data yang besar. Contoh implementasi:

```
main.py X
.vscode > main.py > ...
1 def binary_search(arr, x):
2     low = 0
3     high = len(arr) - 1
4     while low <= high:
5         mid = (low + high) // 2
6         if arr[mid] == x:
7             return mid
8         elif arr[mid] < x:
9             low = mid + 1
10        else:
11            high = mid - 1
12    return -1
13
```

Jump Search

Algoritma ini membagi daftar menjadi blok-blok yang lebih kecil dan melompat ke setiap blok hingga elemen yang dicari diperkirakan berada di dalam blok tertentu. Kemudian pencarian linear dilakukan pada blok tersebut. Jump search memiliki kompleksitas $O(\sqrt{n})$ dan lebih cepat dibanding linear search, tetapi tidak secepat binary search.



```
main.py X
.vscode > main.py > ...
1 import math
2
3 def jump_search(arr, x):
4     n = len(arr)
5     # Menentukan blok lompatan
6     step = math.sqrt(n)
7     prev = 0
8
9     # Melompat hingga menemukan blok di mana elemen target berada
10    while arr[int(min(step, n)-1)] < x:
11        prev = step
12        step += math.sqrt(n)
13        if prev >= n:
14            return -1 # Elemen tidak ditemukan dalam array
15
16    # Melakukan pencarian linear dalam blok tersebut
17    while arr[int(prev)] < x:
18        prev += 1
19        if prev == min(step, n):
20            return -1 # Elemen tidak ditemukan dalam blok
21
22    # Jika elemen ditemukan
23    if arr[int(prev)] == x:
24        return int(prev)
25
26    return -1
27
```

Gambar 1.7

3.4 Algoritma Greedy

Algoritma **greedy** adalah pendekatan algoritmik yang bekerja dengan membuat keputusan optimal pada setiap langkahnya, dengan tujuan mendapatkan hasil terbaik secara keseluruhan. Pada algoritma ini, setiap langkah ditentukan dengan memilih opsi yang tampak paling baik pada saat itu, tanpa mempertimbangkan dampak jangka panjang. Algoritma greedy sering digunakan pada masalah optimasi yang bisa diselesaikan dengan memaksimalkan keuntungan atau meminimalkan biaya.

Greedy berarti "serakah" karena algoritma ini selalu mengambil solusi yang tampak terbaik di setiap tahap, namun tidak selalu menghasilkan solusi optimal global. Hal ini tergantung pada struktur masalahnya. Masalah seperti pemilihan aktivitas, knapsack fractional, dan jalur terpendek bisa diselesaikan dengan algoritma ini.

Cara kerja algoritma greedy

Algoritma greedy bekerja dengan prinsip sederhana: pada setiap langkah, ia memilih opsi terbaik atau paling optimal yang tersedia pada saat itu, tanpa mempertimbangkan efek dari keputusan tersebut di masa depan. Algoritma ini membuat pilihan yang terlihat paling "serakah" atau menguntungkan dalam jangka pendek, berharap pilihan lokal terbaik akan menghasilkan solusi global terbaik.

Langkah-Langkah Kerja Algoritma Greedy:

1. **Inisialisasi Solusi:** Algoritma dimulai dari status awal yang umumnya berupa kumpulan kosong atau titik awal tertentu.

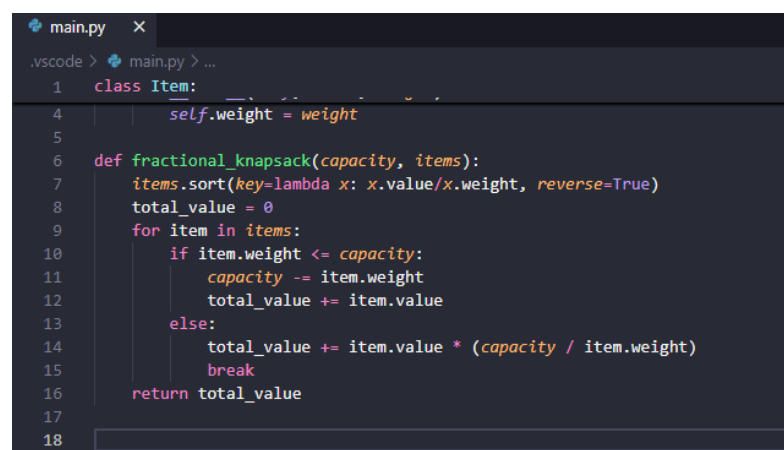
2. **Langkah Pemilihan Optimal:** Pada setiap langkah, algoritma mengevaluasi beberapa pilihan yang mungkin dilakukan. Pilihan terbaik (berdasarkan kriteria yang telah ditentukan) dipilih. Pilihan ini sering kali disebut sebagai **local optimum** atau pilihan terbaik lokal.
3. **Pembaharuan Solusi:** Setelah pilihan terbaik diambil, algoritma memperbarui status solusinya, menambahkan elemen baru ke dalam solusi sementara yang sedang dibangun.
4. **Pengecekan Keadaan Akhir:** Setelah satu langkah diambil, algoritma akan mengecek apakah solusi akhir telah dicapai. Jika kondisi akhir tercapai, algoritma berhenti. Jika belum, algoritma kembali ke langkah pemilihan optimal.
5. **Solusi Akhir:** Setelah menyelesaikan semua langkah, hasil akhir yang dibangun adalah solusi akhir, yang diharapkan merupakan solusi optimal.

Contoh algoritma greedy:

1. Fractional Knapsack Problem

Deskripsi: Diberikan sejumlah barang dengan berat dan nilai masing-masing, dan sebuah knapsack (tas) dengan kapasitas tertentu. Tujuan adalah memaksimalkan nilai barang yang dimasukkan ke dalam knapsack, dengan memungkinkan pengambilan barang secara parsial (sebagian).

Solusi Greedy: Algoritma greedy memilih barang dengan rasio **nilai/berat** tertinggi terlebih dahulu. Barang-barang tersebut dimasukkan ke dalam knapsack sampai kapasitas penuh. Jika ada ruang tersisa tetapi tidak cukup untuk barang berikutnya, sebagian dari barang tersebut diambil.



```

main.py x
.vscode > main.py > ...
1  class Item:
2      def __init__(self, weight, value):
3          self.weight = weight
4          self.value = value
5
6  def fractional_knapsack(capacity, items):
7      items.sort(key=lambda x: x.value/x.weight, reverse=True)
8      total_value = 0
9      for item in items:
10         if item.weight <= capacity:
11             capacity -= item.weight
12             total_value += item.value
13         else:
14             total_value += item.value * (capacity / item.weight)
15             break
16     return total_value
17
18

```

Gambar 1.8 Fractional Knapsack Problem

2. Activity Selection Problem

Deskripsi: Diberikan sejumlah aktivitas dengan waktu mulai dan waktu selesai. Tujuan adalah memilih sebanyak mungkin aktivitas yang dapat dijalankan tanpa ada aktivitas yang bertumpang tindih.

Solusi Greedy: Algoritma greedy memilih aktivitas yang selesai paling awal terlebih dahulu, kemudian memeriksa apakah aktivitas berikutnya tidak bertabrakan dengan yang sudah dipilih.



```
main.py X
.vscode > main.py > ...
1 def activity_selection(start, finish):
2     n = len(finish)
3     print("Aktivitas yang dipilih adalah: ")
4     i = 0
5     print(i, end=" ")
6     for j in range(1, n):
7         if start[j] >= finish[i]:
8             print(j, end=" ")
9             i = j
10
11 # Contoh input
12 start = [1, 3, 0, 5, 8, 5]
13 finish = [2, 4, 6, 7, 9, 9]
14 activity_selection(start, finish)
15
```

Gambar 2.1 Activity Selection Problem

3.5 Algoritma Backtracking

Algoritma backtracking adalah metode penyelesaian masalah yang digunakan untuk menemukan semua (atau beberapa) solusi dari suatu masalah yang dapat dipecahkan dengan cara eksplorasi. Algoritma ini bekerja dengan membangun solusi secara bertahap, menguji setiap kemungkinan pada setiap langkah, dan jika ditemukan bahwa solusi saat ini tidak dapat menyelesaikan masalah, maka algoritma akan "mundur" atau kembali ke langkah sebelumnya untuk mencoba solusi alternatif.

Konsep Dasar Algoritma Backtracking

1. **Pohon Pencarian:** Algoritma backtracking dapat digambarkan sebagai sebuah pohon pencarian, di mana setiap cabang mewakili pilihan yang dapat diambil. Setiap simpul dalam pohon mewakili suatu state dari masalah.
2. **Kondisi Dasar:** Algoritma backtracking akan memeriksa kondisi dasar atau kondisi akhir untuk menentukan apakah solusi telah ditemukan. Jika tidak, algoritma akan melanjutkan untuk menjelajahi lebih jauh.
3. **Mundur (Backtrack):** Jika solusi yang dicapai tidak valid atau tidak memenuhi kriteria yang ditetapkan, algoritma akan mundur untuk mencoba opsi lain. Ini berarti algoritma menghapus pilihan terakhir dan kembali ke state sebelumnya.

Cara Kerja Algoritma Backtracking

1. **Inisialisasi:** Mulai dengan kondisi awal, sering kali dengan tidak ada solusi yang dibangun.
2. **Pengambilan Keputusan:** Pilih opsi yang mungkin dan tambahkan ke solusi sementara.

3. **Cek Validitas:** Setelah setiap keputusan, periksa apakah solusi sementara masih valid. Jika ya, lanjutkan ke langkah berikutnya; jika tidak, kembali ke langkah sebelumnya.
4. **Ulangi:** Ulangi langkah 2 dan 3 sampai solusi ditemukan atau semua opsi telah dieksplorasi.
5. **Solusi Akhir:** Setelah semua kemungkinan dieksplorasi, jika ada solusi yang valid, kembalikan solusi tersebut.

Contoh Algoritma Backtracking

1. **N-Queens:** Dalam contoh N-Queens, fungsi `is_safe` digunakan untuk memeriksa apakah penempatan ratu di posisi tertentu aman. Jika aman, algoritma melanjutkan untuk menempatkan ratu di baris berikutnya. Jika tidak ada solusi yang ditemukan, algoritma mundur untuk mencoba posisi lain.

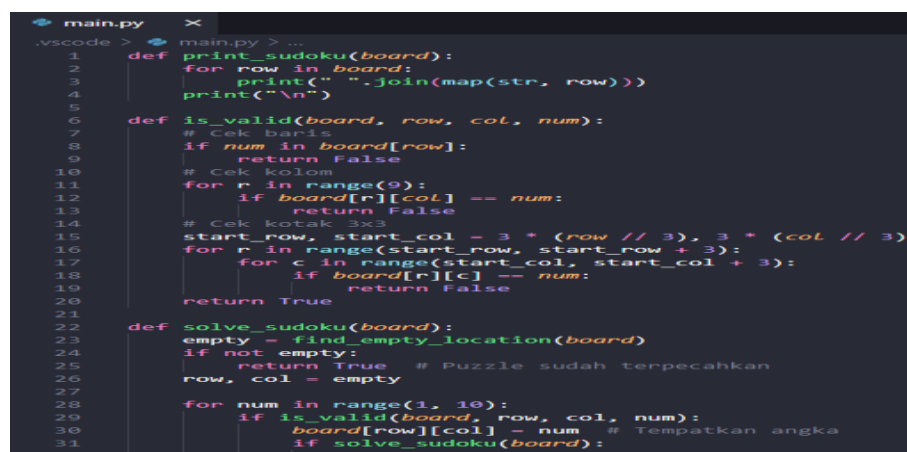


```

main.py X
.vscod... main.py > ...
1 def print_board(board):
2     for row in board:
3         print(" ".join(row))
4     print("\n")
5
6 def is_safe(board, row, col):
7     # Cek kolom
8     for i in range(row):
9         if board[i][col] == 'Q':
10            return False
11
12    # Cek diagonal kiri atas
13    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
14        if board[i][j] == 'Q':
15            return False
16
17    # Cek diagonal kanan atas
18    for i, j in zip(range(row, -1, -1), range(col, len(board))):
19        if board[i][j] == 'Q':
20            return False
21    return True
22
23 def solve_n_queens(board, row):
24     if row == len(board):
25         print_board(board)
26         return True
27
28     for col in range(len(board)):
29         if is_safe(board, row, col):
30             board[row][col] = 'Q' # Tempatkan ratu
31             solve_n_queens(board, row + 1) # Rekursif untuk baris berikutnya
32             board[row][col] = '.' # Mundur (backtrack)
33
34     return False
35
36 def n_queens(n):
37     board = [['.' for _ in range(n)] for _ in range(n)]
38     solve_n_queens(board, 0)
39
40 # Contoh penggunaan
41 n_queens(4)
  
```

Gambar 2.2 N-Queens

2. **Sudoku:** Dalam contoh Sudoku, fungsi `is_valid` memeriksa apakah angka yang akan ditempatkan di posisi kosong sesuai dengan aturan Sudoku. Jika valid, algoritma melanjutkan untuk mengisi posisi berikutnya dan mundur jika tidak ada solusi.



```

main.py X
.vscod... main.py > ...
1 def print_sudoku(board):
2     for row in board:
3         print(" ".join(map(str, row)))
4     print("\n")
5
6 def is_valid(board, row, col, num):
7     # Cek baris
8     if num in board[row]:
9         return False
10
11    # Cek kolom
12    for r in range(9):
13        if board[r][col] == num:
14            return False
15
16    # Cek kotak 3x3
17    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
18    for r in range(start_row, start_row + 3):
19        for c in range(start_col, start_col + 3):
20            if board[r][c] == num:
21                return False
22    return True
23
24 def solve_sudoku(board):
25     empty = find_empty_location(board)
26     if not empty:
27         return True # Puzzle sudah terpecahkan
28
29     row, col = empty
30
31     for num in range(1, 10):
32         if is_valid(board, row, col, num):
33             board[row][col] = num # Tempatkan angka
34             if solve_sudoku(board):
35                 return True # Rekursif
  
```

```

33         board[row][col] = 0 # Mundur (backtrack)
34
35     return False
36
37 def find_empty_location(board):
38     for i in range(9):
39         for j in range(9):
40             if board[i][j] == 0:
41                 return (i, j) # Kembalikan baris dan kolom kosong
42     return None
43
44 # Contoh penggunaan
45 sudoku_board = [
46     [5, 3, 0, 0, 7, 0, 0, 0, 0],
47     [6, 0, 0, 1, 9, 5, 0, 0, 0],
48     [0, 9, 8, 0, 0, 0, 0, 6, 0],
49     [8, 0, 0, 0, 6, 0, 0, 0, 3],
50     [4, 0, 0, 8, 0, 3, 0, 0, 1],
51     [7, 0, 0, 0, 2, 0, 0, 0, 6],
52     [0, 6, 0, 0, 0, 0, 2, 8, 0],
53     [0, 0, 0, 4, 1, 9, 0, 0, 5],
54     [0, 0, 0, 0, 8, 0, 0, 7, 9]
55 ]
56
57 if solve_sudoku(sudoku_board):
58     print_sudoku(sudoku_board)
59 else:
60     print("No solution exists")
61

```

Gambar 2,3 sudoku

Kesimpulan algoritma backtracking

Algoritma ini sering digunakan untuk menyelesaikan berbagai masalah, termasuk tetapi tidak terbatas pada, N-Queens, Sudoku, pemrograman dinamis, dan banyak masalah optimasi lainnya. Sebagai contoh, dalam masalah N-Queens, algoritma backtracking membantu menempatkan ratu di papan catur dengan cara yang memastikan bahwa tidak ada dua ratu yang saling menyerang. Dalam Sudoku, algoritma ini secara efektif memecahkan teka-teki dengan mengisi angka-angka yang valid berdasarkan aturan yang ada.

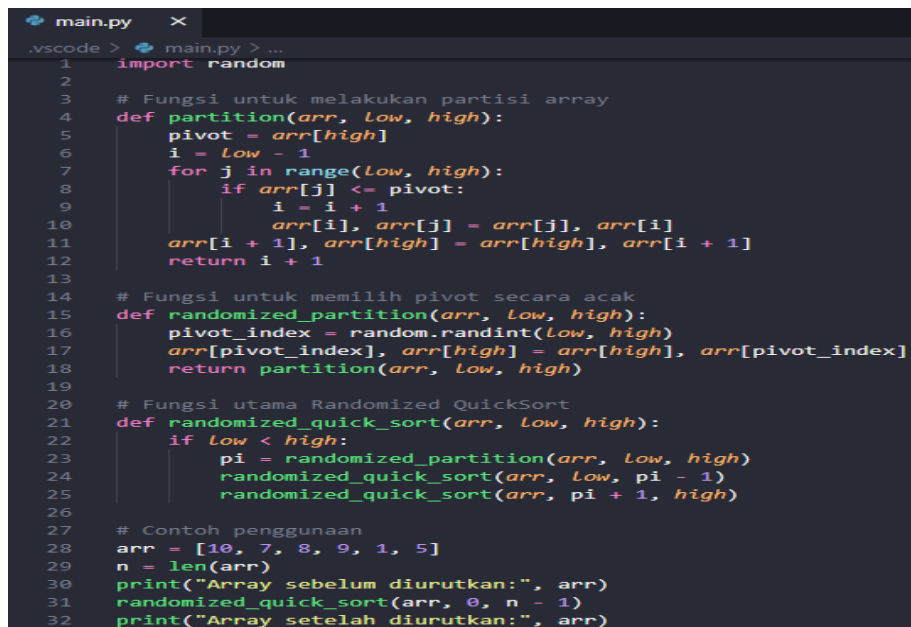
3.6 Algoritma randomized

Algoritma **randomized** adalah algoritma yang memanfaatkan penggunaan elemen **acak** atau **probabilistik** untuk mencapai hasil. Pada setiap eksekusi, algoritma ini menggunakan angka acak dalam prosesnya untuk mempengaruhi jalannya eksekusi. Ini berarti bahwa meskipun diberikan input yang sama, hasil dari algoritma randomized bisa berbeda-beda tergantung pada angka acak yang dihasilkan selama eksekusi. Algoritma ini sering digunakan dalam kasus di mana pendekatan deterministik mungkin terlalu lambat, sulit diimplementasikan, atau terlalu mahal secara komputasional.

Algoritma randomized sering kali lebih sederhana, lebih cepat, dan kadang-kadang bahkan lebih kuat daripada algoritma deterministik. Algoritma ini juga sering digunakan dalam situasi di mana hasil optimal tidak selalu diperlukan, dan penyelesaian yang cukup baik dapat diterima dengan lebih cepat.

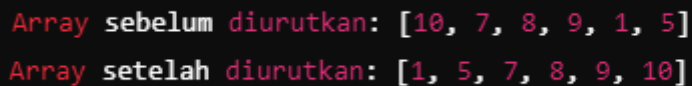
Contoh algoritma randomized

1. **Randomized QuickSort** adalah varian dari algoritma quicksort di mana pivot dipilih secara acak. Hal ini bertujuan untuk mengurangi kemungkinan terjadinya kasus terburuk, terutama jika data input sudah hampir terurut atau terurut dengan pola tertentu. Dengan memilih pivot secara acak, algoritma ini memastikan distribusi yang lebih baik dari data yang akan diproses, sehingga mengurangi kemungkinan terjadinya pembagian yang tidak seimbang.



```
main.py x
.vscode > main.py > ...
1 import random
2
3 # Fungsi untuk melakukan partisi array
4 def partition(arr, low, high):
5     pivot = arr[high]
6     i = low - 1
7     for j in range(low, high):
8         if arr[j] <= pivot:
9             i = i + 1
10            arr[i], arr[j] = arr[j], arr[i]
11    arr[i + 1], arr[high] = arr[high], arr[i + 1]
12    return i + 1
13
14 # Fungsi untuk memilih pivot secara acak
15 def randomized_partition(arr, low, high):
16     pivot_index = random.randint(low, high)
17     arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
18     return partition(arr, low, high)
19
20 # Fungsi utama Randomized QuickSort
21 def randomized_quick_sort(arr, low, high):
22     if low < high:
23         pi = randomized_partition(arr, low, high)
24         randomized_quick_sort(arr, low, pi - 1)
25         randomized_quick_sort(arr, pi + 1, high)
26
27 # Contoh penggunaan
28 arr = [10, 7, 8, 9, 1, 5]
29 n = len(arr)
30 print("Array sebelum diurutkan:", arr)
31 randomized_quick_sort(arr, 0, n - 1)
32 print("Array setelah diurutkan:", arr)
```

Gambar 2.4 Randomized QuickSort



```
Array sebelum diurutkan: [10, 7, 8, 9, 1, 5]
Array setelah diurutkan: [1, 5, 7, 8, 9, 10]
```

Gambar 2.5 output Randomized QuickSort