

ARTIKEL INFORMATIF
JENIS-JENIS ALGORITMA DAN CONTOHNYA



Elvi Yunilta_D0424308

*Disusun untuk memenuhi tugas matakuliah
algoritma dan struktur data*

UNIVERSITAS SULAWESI BARAT

04/okt/2024

ABSTRAK

Algoritma merupakan langkah-langkah logis yang disusun secara sistematis untuk memecahkan suatu masalah. Dalam pemrograman, algoritma menjadi fondasi utama dalam pengembangan perangkat lunak karena menentukan cara kerja dan efisiensi suatu program. Terdapat berbagai jenis algoritma yang digunakan dalam berbagai konteks pemrograman. Artikel ini membahas beberapa jenis algoritma umum, termasuk **algoritma rekursif**, **algoritma sorting**, **algoritma searching**, **algoritma greedy**, **algoritma backtracking**, dan **algoritma randomized**. Setiap jenis algoritma memiliki karakteristik dan cara kerja yang berbeda, serta efisiensi yang bergantung pada kasus spesifik. Sebagai contoh, algoritma **Merge Sort** merupakan salah satu algoritma sorting yang menggunakan pendekatan divide-and-conquer, sementara **algoritma greedy** memecahkan masalah dengan memilih solusi terbaik lokal dalam setiap langkah. Pembahasan ini juga menyertakan contoh implementasi dari masing **Kata Kunci**: algoritma, pemrograman, sorting, searching, rekursif, greedy, backtracking, randomized-masing algoritma untuk menunjukkan cara penggunaannya dalam memecahkan berbagai masalah pemrograman.

Kata Kunci: algoritma, pemrograman, sorting, searching, rekursif, greedy, backtracking, randomized

PENDAHULUAN

Algoritma merupakan inti dari proses komputasi dalam pemrograman. Setiap program komputer pada dasarnya adalah implementasi dari satu atau lebih algoritma, yang didefinisikan sebagai serangkaian langkah logis dan sistematis untuk memecahkan suatu masalah atau mencapai tujuan tertentu. Algoritma sangat penting karena menentukan cara komputer memproses data dan menyelesaikan tugas dengan efisien.

Dalam dunia pemrograman, terdapat berbagai jenis algoritma yang masing-masing memiliki fungsi dan karakteristik khusus. Pemahaman tentang berbagai jenis algoritma sangat penting bagi pengembang perangkat lunak untuk memilih solusi yang paling tepat dan efisien dalam menghadapi berbagai permasalahan. Beberapa algoritma berfokus pada pengurutan data, seperti **algoritma sorting**; sementara lainnya bertujuan untuk pencarian data, seperti **algoritma searching**. Selain itu, terdapat juga algoritma yang mengutamakan pendekatan optimal dalam setiap langkah, seperti **algoritma greedy**, atau yang berupaya menyelesaikan masalah dengan mencoba semua kemungkinan solusi, seperti **algoritma backtracking**.

Pendahuluan ini bertujuan untuk memberikan gambaran umum tentang berbagai jenis algoritma yang sering digunakan dalam pemrograman beserta contoh implementasinya. Setiap algoritma dibahas secara mendalam untuk memberikan pemahaman lebih baik tentang cara kerja, keunggulan, dan kekurangannya.

METODE

Metode yang digunakan dalam menganalisis atau membuat artikel ini diantaranya studi literatur, di mana penulis mengumpulkan informasi dari berbagai sumber, termasuk buku, artikel, dan sumber daring terkait jenis algoritma contohnya dan penerapannya dalam pemrograman.

PEMBAHASAN

1.1 JENIS-JENIS ALGORITMA

2.1 Algoritma Rekursif

Algoritma rekursif adalah teknik pemrograman yang menggunakan fungsi atau prosedur yang memanggil dirinya sendiri untuk menyelesaikan masalah. Dalam konteks ini, "rekursif" mengacu pada proses berulang yang terjadi ketika fungsi memanggil dirinya sendiri.

Bayangkan seperti ini: Anda memiliki teka-teki yang kompleks. Anda memecah teka-teki itu menjadi bagian-bagian yang lebih kecil, dan kemudian memecah bagian-bagian itu menjadi bagian yang lebih kecil lagi. Anda terus melakukan ini sampai Anda memiliki bagian-bagian yang sangat kecil yang mudah diselesaikan. Kemudian, Anda menggabungkan solusi dari bagian-bagian kecil ini untuk menyelesaikan teka-teki besar.

Algoritma rekursif bekerja dengan cara yang sama. Fungsi rekursif memecah masalah menjadi sub-masalah yang lebih kecil, dan kemudian memanggil dirinya sendiri untuk menyelesaikan sub-masalah ini. Proses ini berlanjut sampai masalah dasar tercapai, yang dapat diselesaikan secara langsung. Kemudian, solusi dari sub-masalah dikombinasikan untuk menghasilkan solusi untuk masalah utama.

Cara Kerja Algoritma rekursif

1. Kasus Dasar (Base Case): Ini adalah kondisi yang menghentikan rekursi. Ketika kasus dasar tercapai, fungsi tidak memanggil dirinya sendiri lagi dan mengembalikan nilai. Kasus dasar sangat penting untuk mencegah rekursi tak terbatas, yang dapat menyebabkan kesalahan pemrograman.

2. Langkah Rekursif (Recursive Step): Ini adalah bagian dari fungsi yang memanggil dirinya sendiri dengan versi masalah yang lebih kecil. Langkah rekursif harus dirancang untuk mendekati kasus dasar secara bertahap.

Teori Algoritma Rekursif

Algoritma rekursif didasarkan pada prinsip induksi matematika. Induksi matematika adalah metode pembuktian yang digunakan untuk membuktikan pernyataan tentang bilangan bulat. Prinsip induksi matematika menyatakan bahwa untuk membuktikan bahwa pernyataan benar untuk semua bilangan bulat n yang lebih besar dari atau sama dengan k , kita perlu melakukan dua hal:

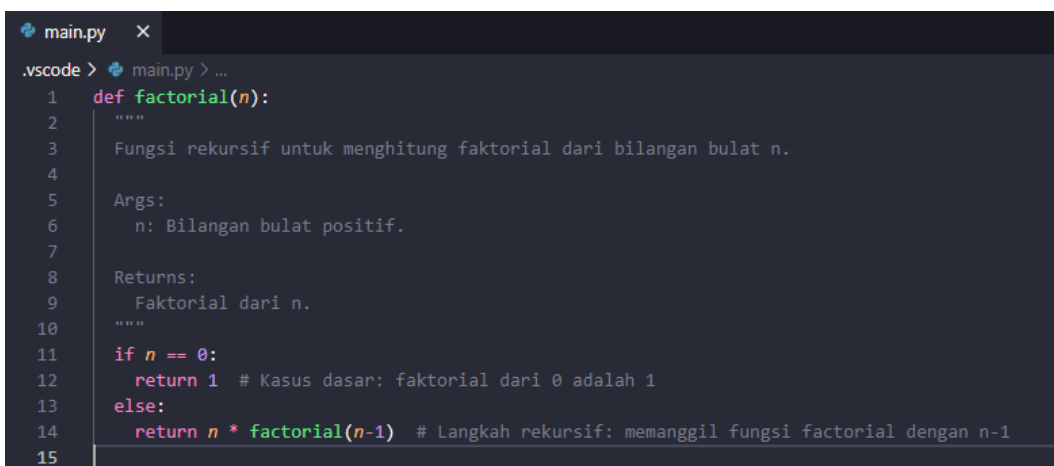
1. **Kasus Dasar:** Buktikan bahwa pernyataan benar untuk $n = k$.
2. **Langkah Induktif:** Asumsikan bahwa pernyataan benar untuk beberapa bilangan bulat $n = m$, dan buktikan bahwa pernyataan juga benar untuk $n = m + 1$.

Algoritma rekursif mirip dengan induksi matematika karena juga memiliki kasus dasar dan langkah induktif. Kasus dasar adalah kondisi yang menghentikan rekursi, dan langkah rekursif adalah langkah yang mengasumsikan bahwa solusi untuk masalah yang lebih kecil sudah tersedia dan menggunakannya untuk menyelesaikan masalah yang lebih besar.

Contoh Algoritma Rekursif: Menghitung Faktorial

Faktorial dari bilangan bulat positif n didefinisikan sebagai perkalian dari semua bilangan bulat positif dari 1 hingga n . Misalnya, faktorial dari 5 (ditulis sebagai $5!$) adalah $5 * 4 * 3 * 2 * 1 = 120$.

Berikut adalah contoh implementasi fungsi rekursif untuk menghitung faktorial dalam bahasa Python:



```
main.py x
.vscode > main.py > ...
1 def factorial(n):
2     """
3     Fungsi rekursif untuk menghitung faktorial dari bilangan bulat n.
4
5     Args:
6         n: Bilangan bulat positif.
7
8     Returns:
9         Faktorial dari n.
10    """
11    if n == 0:
12        return 1 # Kasus dasar: faktorial dari 0 adalah 1
13    else:
14        return n * factorial(n-1) # Langkah rekursif: memanggil fungsi factorial dengan n-1
15
```

Gambar 1.1 contoh algoritma rekursif

Penjelasan:

- Kasus Dasar: Jika n sama dengan 0, fungsi mengembalikan 1. Ini adalah kasus dasar yang menghentikan rekursi.
- Langkah Rekursif: Jika n lebih besar dari 0, fungsi memanggil dirinya sendiri dengan $n-1$ sebagai argumen. Hasil dari panggilan rekursif ini dikalikan dengan n untuk mendapatkan faktorial dari n .

Keuntungan dan Kerugian Algoritma Rekursif

Keuntungan:

- Kode yang lebih ringkas dan mudah dibaca: Algoritma rekursif dapat membuat kode lebih mudah dipahami dan ditulis, terutama untuk masalah yang kompleks.
- Solusi elegan untuk masalah yang dapat dipecah menjadi sub-masalah yang serupa: Algoritma rekursif sangat cocok untuk masalah yang dapat dipecah menjadi versi yang lebih kecil dari masalah itu sendiri.
- Mudah untuk diimplementasikan: Algoritma rekursif relatif mudah untuk diimplementasikan, terutama jika Anda memahami konsep dasar rekursi.

Kerugian:

- Memori yang lebih banyak digunakan: Algoritma rekursif dapat menggunakan lebih banyak memori daripada algoritma iteratif, karena setiap panggilan rekursif menempati ruang di stack.
- Dapat sulit untuk di-debug: Algoritma rekursif dapat lebih sulit untuk di-debug daripada algoritma iteratif, karena aliran kontrol lebih kompleks dan lebih sulit untuk diikuti.
- Performa yang lebih lambat: Algoritma rekursif dapat lebih lambat daripada algoritma iteratif, terutama jika kedalaman rekursi sangat besar.

2.2 algoritma sorting

Algoritma sorting adalah metode atau teknik untuk mengurutkan data atau elemen-elemen dalam suatu struktur data secara terstruktur, berdasarkan aturan tertentu. Tujuan utama dari proses sorting adalah untuk mengubah data yang tidak teratur menjadi urutan yang teratur, misalnya dari data yang tidak teratur menjadi data yang teratur menaik atau menurun. [1] [2]

Jenis-jenis Algoritma Sorting

Terdapat berbagai jenis algoritma sorting, masing-masing dengan kelebihan dan kekurangannya sendiri. Berikut beberapa jenis algoritma sorting yang umum digunakan:

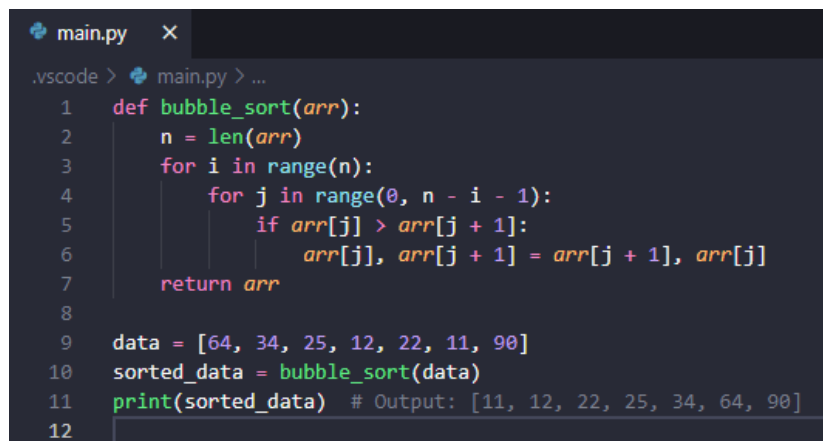
1. Bubble Sort:

- Merupakan salah satu algoritma sorting yang paling sederhana.
- Bekerja dengan cara membandingkan setiap pasangan elemen yang berdekatan dan menukarnya jika tidak dalam urutan yang diinginkan.
- Proses ini diulang sampai seluruh array telah diperiksa dan tidak ada pertukaran lagi yang bisa dilakukan. [1]
- Cocok untuk data yang hampir terurut.

Teori

Bubble Sort menggunakan pendekatan iterasi untuk mengurutkan data. Pada setiap iterasi, algoritma membandingkan elemen yang berdekatan dan menukarnya jika tidak dalam urutan yang diinginkan. Proses ini dilakukan berulang kali hingga tidak ada lagi pertukaran yang diperlukan.

Contoh



```
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      return arr
8
9  data = [64, 34, 25, 12, 22, 11, 90]
10 sorted_data = bubble_sort(data)
11 print(sorted_data) # Output: [11, 12, 22, 25, 34, 64, 90]
12
```

Gambar 1.2 bubble sort

2. Merge sort

Merge sort adalah algoritma pengurutan yang sangat efisien yang mengikuti strategi bagi-dan-kuasai. Algoritma ini memecah daftar yang tidak diurutkan menjadi subdaftar yang lebih kecil, mengurutkan setiap subdaftar secara rekursif, lalu menggabungkan kembali subdaftar yang telah diurutkan untuk membentuk daftar akhir yang diurutkan.

Teori:

- **Membagi:** Daftar dibagi menjadi dua bagian secara rekursif hingga setiap subdaftar hanya berisi satu elemen (yang secara inheren diurutkan).
- **Conquer:** Setiap sublist diurutkan secara individual. Karena setiap sublist hanya memiliki satu elemen, maka sublist tersebut sudah diurutkan.

- **Gabungkan:** Subdaftar yang telah diurutkan digabungkan kembali dengan cara yang telah diurutkan. Proses penggabungan ini sangat penting bagi efisiensi algoritma.

Contoh:



```

main.py
.vscode > CAUsers\acero\latihan python\vscode\main.py
1 def merge_sort(arr):
2     """
3     Sorts a list using the merge sort algorithm.
4
5     Args:
6         arr: The list to be sorted.
7
8     Returns:
9         The sorted list.
10    """
11    if len(arr) > 1:
12        mid = len(arr) // 2 # Find the middle point and divide the list
13        left_half = arr[:mid]
14        right_half = arr[mid:]
15        merge_sort(left_half) # Recursively sort the left half
16        merge_sort(right_half) # Recursively sort the right half
17
18    i = j = k = 0
19    # Merge the sorted halves into a single sorted list
20    while i < len(left_half) and j < len(right_half):
21        if left_half[i] < right_half[j]:
22            arr[k] = left_half[i]
23            i += 1
24        else:
25            arr[k] = right_half[j]
26            j += 1
27        k += 1
28    # Copy the remaining elements of the left half, if any
29    while i < len(left_half):
30        arr[k] = left_half[i]
31        i += 1
32        k += 1
33
34    # Copy the remaining elements of the right half, if any
35    while j < len(right_half):
36        arr[k] = right_half[j]
37        j += 1
38        k += 1
39
40    # Example usage
41    test_list = [8, 3, 1, 7, 0, 10, 2]
42    merge_sort(test_list)
43    print(test_list) # Output: [0, 1, 2, 3, 7, 8, 10]

```

Gambar 1.3 marge sort

Penjelasan:

1. **merge_sort(arr)fungsi:**
 - Mengambil daftar yang tidak diurutkan arrsebagai input.
 - **Kasus dasar:** Jika daftar hanya mempunyai satu elemen (len(arr) <= 1), daftar tersebut sudah diurutkan, sehingga fungsi akan mengembalikannya.
 - **Membagi:** Menemukan indeks tengah (mid) dan membagi daftar menjadi dua bagian: left_halfdan right_half.
 - **Conquer:** Memanggil kedua bagian secara rekursif merge_sortuntuk mengurutkannya secara independen.
 - **Gabungkan:** Memanggil mergefungsi (tidak ditampilkan dalam contoh ini) untuk menggabungkan bagian-bagian yang diurutkan menjadi satu daftar yang diurutkan.

2. mergefungsi (tidak ditampilkan):

- Mengambil dua daftar yang diurutkan sebagai input.
- Membandingkan elemen dari kedua daftar, menyalin elemen yang lebih kecil ke daftar keluaran.
- Berlanjut hingga salah satu daftar masukan habis.
- Menyalin elemen yang tersisa dari daftar yang belum habis ke daftar keluaran.

Poin Utama:

- **Rekursif:** Merge sort adalah algoritma rekursif, yang berarti ia memanggil dirinya sendiri berulang kali untuk memecahkan submasalah yang lebih kecil.
- **Stabil:** Merge sort merupakan algoritma pengurutan yang stabil, artinya ia mempertahankan urutan relatif dari elemen yang sama.
- **Kompleksitas Waktu:** $O(n \log n)$ untuk semua kasus, membuatnya sangat efisien untuk kumpulan data besar.
- **Kompleksitas Ruang:** $O(n)$ karena ruang tambahan diperlukan untuk penggabungan.

3. Quick sort

Quick sort adalah algoritma pengurutan efisien lainnya yang juga menggunakan strategi bagi-dan-kuasai. Ia membagi array input di sekitar elemen pivot, menempatkan semua elemen yang lebih kecil dari pivot di sebelah kirinya dan semua elemen yang lebih besar dari pivot di sebelah kanannya. Proses pembagian ini kemudian diterapkan secara rekursif ke subarray hingga seluruh array diurutkan.

Teori:

- **Membagi:** Array input dipartisi di sekitar elemen pivot. Ini berarti bahwa elemen yang lebih kecil dari pivot ditempatkan di sebelah kirinya, dan elemen yang lebih besar dari pivot ditempatkan di sebelah kanannya.
- **Conquer:** Dua subarray (kiri dan kanan pivot) diurutkan secara rekursif menggunakan quick sort.
- **Gabungkan:** Karena langkah partisi sudah menyusun elemen relatif terhadap pivot, tidak diperlukan langkah penggabungan eksplisit. Susunan diurutkan di tempat.

Algoritma:

1. **Pilih titik tumpu:** Pilih elemen titik tumpu dari array input. Pilihan umum meliputi elemen pertama, terakhir, atau tengah.
2. **Partisi:** Susun ulang array sedemikian rupa sehingga semua elemen yang lebih kecil dari pivot ditempatkan di depannya, dan semua elemen yang lebih besar

dari pivot ditempatkan setelahnya. Posisi akhir pivot adalah posisi yang diurutkan dengan benar.

3. **Urutkan secara rekursif:** Terapkan pengurutan cepat secara rekursif ke subarray di kedua sisi pivot.

Contoh:

```
1 def quick_sort(arr, low, high):
2     """
3     Sorts a list using the quick sort algorithm.
4
5     Args:
6         arr: The list to be sorted.
7         low: The starting index of the subarray.
8         high: The ending index of the subarray.
9     """
10    if low < high:
11        pi = partition(arr, low, high) # Partition the array
12        quick_sort(arr, low, pi - 1) # Recursively sort left subarray
13        quick_sort(arr, pi + 1, high) # Recursively sort right subarray
14
15    def partition(arr, low, high):
16        """
17        Partitions the array around a pivot element.
18
19        Args:
20            arr: The list to be partitioned.
21            low: The starting index of the subarray.
22            high: The ending index of the subarray.
23
24        Returns:
25            The index of the pivot element after partitioning.
26        """
27        pivot = arr[high] # Choosing the last element as the pivot
28        i = low - 1
29
30        for j in range(low, high):
31            if arr[j] <= pivot:
32                i += 1
33                arr[i], arr[j] = arr[j], arr[i] # Swap elements
34
35        arr[i + 1], arr[high] = arr[high], arr[i + 1] # Swap pivot with the element at i+1
36        return i + 1
37
38    # Example usage
39    test_list = [8, 3, 1, 7, 0, 10, 2]
40    quick_sort(test_list, 0, len(test_list) - 1)
41    print(test_list) # Output: [0, 1, 2, 3, 7, 8, 10]
```

Gambar 1.4 Quick sort

Penjelasan:

1. quick_sort(arr, low, high)fungsi:

- Mengambil daftar yang tidak diurutkan arr, indeks awal low, dan indeks akhir high sebagai input.
- **Kasus dasar:** Jika low lebih besar dari atau sama dengan high, subarray sudah diurutkan, sehingga fungsi kembali.
- **Membagi (Partisi):** Memanggil partition fungsi untuk mempartisi subarray di sekitar pivot.
- **Conquer:** quick_sort Memanggil subarray kiri (low to pi - 1) dan subarray kanan (pi + 1 to) secara rekursif high.

2. partition(arr, low, high)fungsi:

- Mengambil daftar arr, indeks awal low, dan indeks akhir high sebagai input.
- Memilih elemen terakhir sebagai pivot.

- Beriterasi melalui subarray, menukar elemen yang lebih kecil dari pivot ke sisi kiri.
- Menukar pivot dengan elemen di $i + 1$, menempatkan pivot pada posisi yang diurutkan dengan benar.
- Mengembalikan indeks pivot setelah partisi.

2.3 Algoritma Searching

Algoritme pencarian merupakan hal mendasar bagi ilmu komputer, yang memungkinkan kita menemukan data tertentu secara efisien dalam suatu koleksi. Algoritme ini penting untuk tugas-tugas seperti menemukan informasi dalam basis data, mengambil file dari sistem file, dan melakukan pencocokan pola dalam pemrosesan teks. Panduan ini memberikan gambaran umum yang komprehensif tentang algoritme pencarian umum, karakteristiknya, dan aplikasinya.

Jenis-jenis algoritma searching:

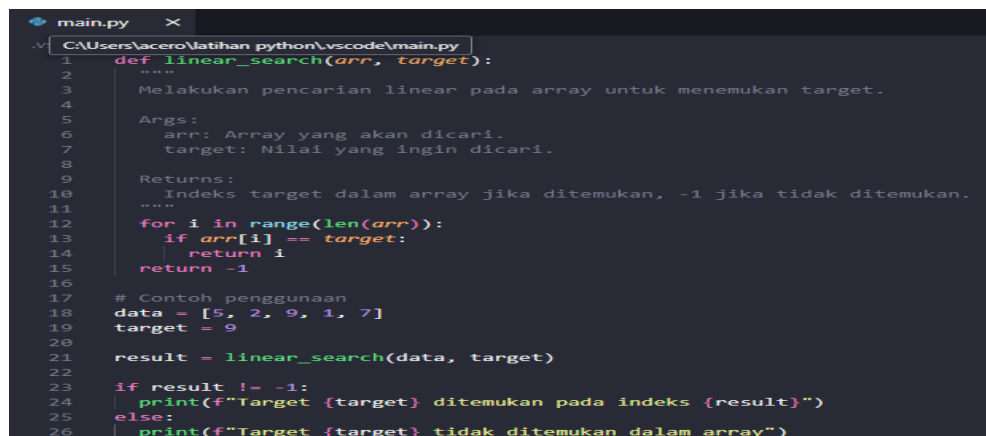
1. Linear search

Pencarian linear adalah algoritma pencarian yang paling dasar. Ia bekerja dengan memeriksa setiap elemen dalam kumpulan data secara berurutan, membandingkannya dengan nilai target hingga ditemukan atau sampai seluruh kumpulan data telah diperiksa.

Teori:

- **Konsep:** Algoritma ini memeriksa setiap elemen dalam kumpulan data satu per satu, membandingkannya dengan nilai target. Jika nilai target ditemukan, pencarian berhenti, dan posisi elemen tersebut dikembalikan. Jika tidak ditemukan, pencarian akan terus berlanjut hingga akhir kumpulan data.
- **Kompleksitas Waktu:**
 - Kasus Terbaik: $O(1)$ (jika target adalah elemen pertama)
 - Kasus Rata-rata dan Terburuk: $O(n)$ (jika target berada di akhir atau tidak ada)
- **Kompleksitas Ruang:** $O(1)$

Contoh:



```

1  def linear_search(arr, target):
2      """
3      Melakukan pencarian linear pada array untuk menemukan target.
4
5      Args:
6          arr: Array yang akan dicari.
7          target: Nilai yang ingin dicari.
8
9      Returns:
10         Indeks target dalam array jika ditemukan, -1 jika tidak ditemukan.
11     """
12     for i in range(len(arr)):
13         if arr[i] == target:
14             return i
15     return -1
16
17 # Contoh penggunaan
18 data = [5, 2, 9, 1, 7]
19 target = 9
20
21 result = linear_search(data, target)
22
23 if result != -1:
24     print(f"Target {target} ditemukan pada indeks {result}")
25 else:
26     print(f"Target {target} tidak ditemukan dalam array")
  
```

Gambar 1.5 linear search

Pembahasan:

1. **linear_search(arr, target):** Fungsi ini menerima array arr dan nilai target target sebagai input.
2. **for i in range(len(arr)):** Loop ini iterasi melalui setiap elemen dalam array.
3. **if arr[i] == target:** Kondisi ini memeriksa apakah elemen saat ini sama dengan nilai target.
4. **return i:** Jika target ditemukan, fungsi mengembalikan indeks elemen tersebut.
5. **return -1:** Jika target tidak ditemukan setelah memeriksa semua elemen, fungsi mengembalikan -1.

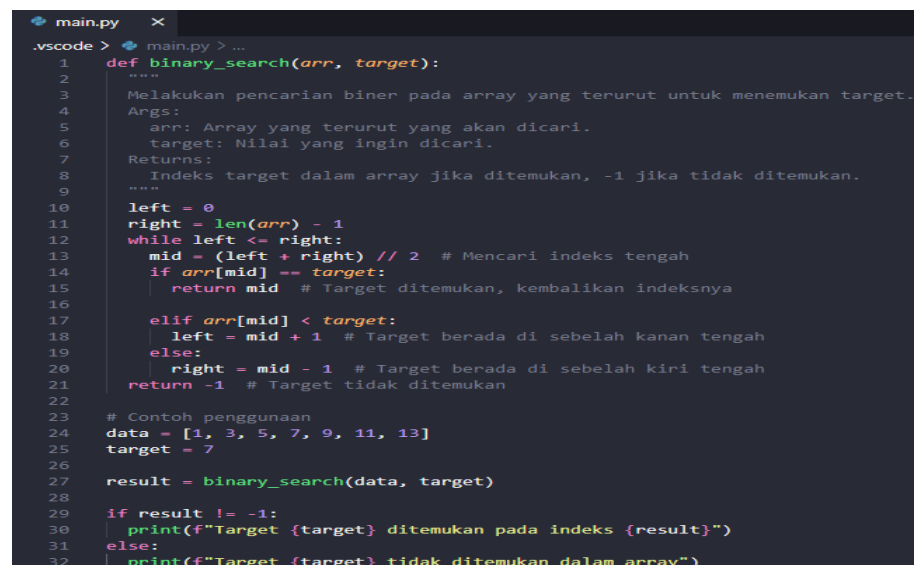
2. Binary search

Pencarian biner adalah algoritma pencarian yang lebih efisien dibandingkan pencarian linear. Ia dirancang khusus untuk kumpulan data yang terurut dan bekerja dengan membagi interval pencarian menjadi dua bagian secara berulang, membuang setengah dari elemen yang tersisa di setiap langkah.

Teori:

- **Konsep:** Algoritma ini bekerja dengan membagi interval pencarian menjadi dua bagian secara berulang. Pada setiap langkah, elemen tengah interval diperiksa. Jika elemen tengah sama dengan nilai target, pencarian selesai. Jika nilai target lebih kecil dari elemen tengah, pencarian dilanjutkan pada bagian kiri interval. Jika nilai target lebih besar dari elemen tengah, pencarian dilanjutkan pada bagian kanan interval. Proses ini berulang hingga nilai target ditemukan atau interval pencarian menjadi kosong.
- **Kompleksitas Waktu:** $O(\log n)$
- **Kompleksitas Ruang:** $O(1)$

Contoh:



```
main.py
.vscode
def binary_search(arr, target):
    """
    Melakukan pencarian biner pada array yang terurut untuk menemukan target.
    Args:
        arr: Array yang terurut yang akan dicari.
        target: Nilai yang ingin dicari.
    Returns:
        Indeks target dalam array jika ditemukan, -1 jika tidak ditemukan.
    """
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2 # Mencari indeks tengah
        if arr[mid] == target:
            return mid # Target ditemukan, kembalikan indeksnya
        elif arr[mid] < target:
            left = mid + 1 # Target berada di sebelah kanan tengah
        else:
            right = mid - 1 # Target berada di sebelah kiri tengah
    return -1 # Target tidak ditemukan

# Contoh penggunaan
data = [1, 3, 5, 7, 9, 11, 13]
target = 7
result = binary_search(data, target)
if result != -1:
    print(f"Target {target} ditemukan pada indeks {result}")
else:
    print(f"Target {target} tidak ditemukan dalam array")
```

Gambar 1.6 binary search

Penjelasan:

1. **binary_search(arr, target):** Fungsi ini menerima array terurut arr dan nilai target target sebagai input.
2. **left = 0 dan right = len(arr) - 1:** Variabel left dan right menginisialisasi batas kiri dan kanan interval pencarian.
3. **while left <= right:** Loop ini berlanjut selama batas kiri masih lebih kecil atau sama dengan batas kanan.
4. **mid = (left + right) // 2:** Variabel mid menghitung indeks tengah interval pencarian.
5. **if arr[mid] == target:** Jika elemen tengah sama dengan nilai target, pencarian selesai, dan fungsi mengembalikan indeks tengah.
6. **elif arr[mid] < target:** Jika elemen tengah lebih kecil dari target, pencarian dilanjutkan pada bagian kanan interval pencarian. Batas kiri diubah menjadi mid + 1 untuk mengecualikan elemen tengah.
7. **else:** Jika elemen tengah lebih besar dari target, pencarian dilanjutkan pada bagian kiri interval pencarian. Batas kanan diubah menjadi mid - 1 untuk mengecualikan elemen tengah.
8. **return -1:** Jika loop selesai tanpa menemukan target, fungsi mengembalikan -1, menunjukkan bahwa target tidak ditemukan dalam array.

3. Hashing

Pencarian hashing adalah teknik pencarian yang sangat efisien yang menggunakan fungsi hash untuk memetakan kunci ke indeks dalam tabel hash. Ini memungkinkan pencarian waktu konstan rata-rata, menjadikannya pilihan yang sangat baik untuk menyimpan dan mengambil data dengan cepat.

Teori:

- **Konsep:** Pencarian hashing bekerja dengan menggunakan fungsi hash, yang memetakan kunci (misalnya, string, angka) ke indeks dalam tabel hash. Tabel hash adalah array yang menyimpan data dan diakses dengan menggunakan fungsi hash. Ketika kita ingin mencari nilai yang terkait dengan kunci tertentu, kita menghitung hash-nya dan menggunakannya untuk mengakses indeks yang sesuai dalam tabel hash.
- **Kompleksitas Waktu:** $O(1)$ (rata-rata)
- **Kompleksitas Ruang:** $O(n)$ (untuk tabel hash)

Contoh:

A screenshot of a code editor window titled 'main.py'. The code defines a 'HashTable' class with several methods: __init__(self, size), __len__(self), __contains__(self, key), insert(self, key, value), get(self, key), and hash(self, key). The __contains__ method uses a loop to check if a key exists in the table. The insert method appends a key-value pair to the table. The get method returns the value for a given key. The hash method returns the hash of a key.

```
1 class HashTable:
2     def __init__(self, size):
3         self.size = size
4         self.table = [None] * size
5
6     def __len__(self):
7         return self.size
8
9     def __contains__(self, key):
10        index = self.hash(key) % self.size
11        if self.table[index] is not None:
12            for k, _ in self.table[index]:
13                if k == key:
14                    return True
15            return False
16        def insert(self, key, value):
17            index = self.hash(key) % self.size
18            if self.table[index] is None:
19                self.table[index] = [(key, value)]
20            else:
21                self.table[index].append((key, value))
22
23        def get(self, key):
24            index = self.hash(key) % self.size
25            if self.table[index] is not None:
26                for k, v in self.table[index]:
27                    if k == key:
28                        return v
29            return None
30
31        def hash(self, key):
32            return hash(key)
```

Gambar 1.7 hashing

Penjelasan:

1. **HashTable Class:** Kelas ini mendefinisikan tabel hash.
2. **__init__(self, size):** Inisialisasi tabel hash dengan ukuran yang ditentukan.
3. **__len__(self):** Mengembalikan ukuran tabel hash.
4. **__contains__(self, key):** Mengecek apakah kunci ada dalam tabel hash.
5. **insert(self, key, value):** Menyimpan pasangan kunci-nilai ke dalam tabel hash.
6. **get(self, key):** Mengambil nilai yang terkait dengan kunci tertentu.
7. **hash(self, key):** Menghitung hash dari kunci.

2.4 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.

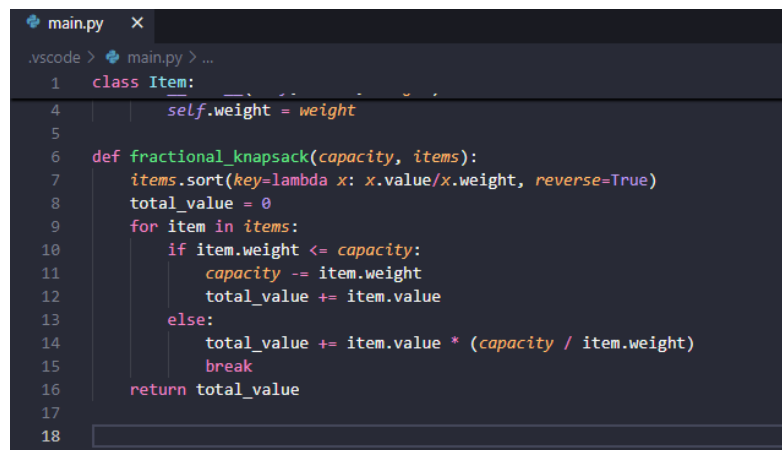
Greedy berarti "serakah" karena algoritma ini selalu mengambil solusi yang tampak terbaik di setiap tahap, namun tidak selalu menghasilkan solusi optimal global. Hal ini tergantung pada struktur masalahnya. Masalah seperti pemilihan aktivitas, knapsack fractional, dan jalur terpendek bisa diselesaikan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

Jenis-jenis 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.



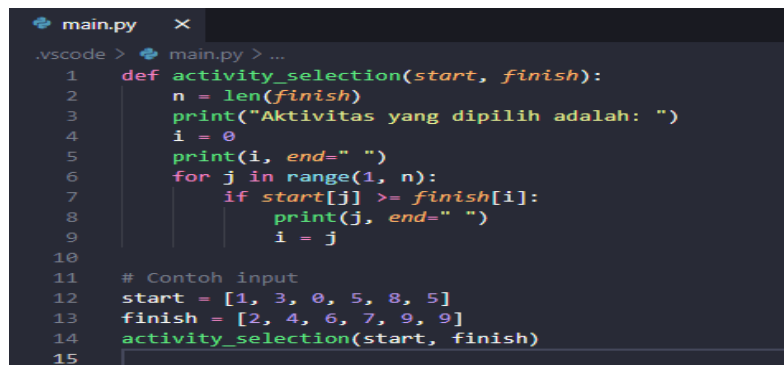
```
main.py X
.vscode > main.py > ...
1 class Item:
2     def __init__(self, weight, value):
3         self.weight = weight
4         self.value = value
5
6 def fractional_knapsack(capacity, items):
7     items.sort(key=lambda x: x.value/x.weight, reverse=True)
8     total_value = 0
9     for item in 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 1.9 Activity Selection Problem

2.5 Algoritma Backtracking

Algoritma backtracking adalah teknik pemecahan masalah yang sistematis untuk menemukan solusi dari suatu masalah dengan menjelajahi semua kemungkinan solusi secara bertahap. Ia bekerja dengan membangun solusi secara bertahap, dan jika solusi saat ini tidak mengarah pada solusi yang valid, algoritma akan "mundur" (backtracking) ke langkah sebelumnya dan mencoba pilihan lain.

Jenis-jenis backtracking

1. N-Queens

Algoritma backtracking adalah teknik pencarian sistematis yang digunakan untuk menemukan solusi untuk masalah dengan menjelajahi semua kemungkinan solusi secara bertahap. Ia bekerja dengan membangun solusi secara bertahap, dan jika solusi saat ini tidak mengarah pada solusi yang valid, algoritma akan "mundur" (backtracking) ke langkah sebelumnya dan mencoba pilihan lain.

Teori:

Masalah N-Queens adalah teka-teki klasik dalam ilmu komputer yang melibatkan penempatan N ratu pada papan catur berukuran $N \times N$ sedemikian rupa sehingga tidak ada dua ratu yang saling menyerang. Ratu dapat menyerang secara horizontal, vertikal, dan diagonal.

1. Konsep Dasar:

- **Papan Catur:** Masalah ini melibatkan papan catur standar berukuran $N \times N$, di mana N adalah jumlah ratu yang akan ditempatkan.
- **Ratu:** Ratu adalah bidak catur yang dapat bergerak dalam jumlah kotak yang tidak terbatas secara horizontal, vertikal, dan diagonal.
- **Aturan Penyerangan:** Dua ratu saling menyerang jika mereka berada pada baris, kolom, atau diagonal yang sama.
- **Tujuan:** Tujuannya adalah untuk menemukan semua kemungkinan penempatan N ratu pada papan catur sehingga tidak ada dua ratu yang saling menyerang.

Contoh:

```
main.py X
.vscode > 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     # Cek diagonal kiri atas
12     for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
13         if board[i][j] == 'Q':
14            return False
15     # Cek diagonal kanan atas
16     for i, j in zip(range(row, -1, -1), range(col, len(board))):
17         if board[i][j] == 'Q':
18            return False
19     return True
20
21 def solve_n_queens(board, row):
22     if row >= len(board):
23         print_board(board)
24         return True
25
26     for col in range(len(board)):
27         if is_safe(board, row, col):
28             board[row][col] = 'Q' # Tempatkan ratu
29             solve_n_queens(board, row + 1) # Rekursif untuk baris berikutnya
30             board[row][col] = '.' # Mundur (backtrack)
31
32     return False
33
34 def n_queens(n):
35     board = [['.' for _ in range(n)] for _ in range(n)]
36     solve_n_queens(board, 0)
```

2. sudoku

Sudoku adalah teka-teki angka yang populer yang melibatkan mengisi grid 9x9 dengan angka dari 1 hingga 9, dengan aturan bahwa setiap baris, kolom, dan kotak 3x3 harus berisi semua angka dari 1 hingga 9 tanpa pengulangan.

Teori Sudoku:

- **Grid:** Sudoku dimainkan pada grid 9x9 yang dibagi menjadi sembilan kotak 3x3.
- **Angka:** Angka yang digunakan dalam Sudoku adalah dari 1 hingga 9.
- **Aturan:**
 - Setiap baris harus berisi semua angka dari 1 hingga 9 tanpa pengulangan.
 - Setiap kolom harus berisi semua angka dari 1 hingga 9 tanpa pengulangan.
 - Setiap kotak 3x3 harus berisi semua angka dari 1 hingga 9 tanpa pengulangan.
- **Tujuan:** Tujuannya adalah untuk mengisi semua kotak kosong dalam grid dengan angka yang valid, mengikuti aturan Sudoku.

Algoritma Backtracking untuk Memecahkan Sudoku:

Algoritma backtracking adalah pendekatan umum yang digunakan untuk memecahkan teka-teki Sudoku. Algoritma ini bekerja dengan mencoba mengisi setiap kotak kosong dengan angka yang valid secara bergantian. Jika angka tersebut valid (tidak melanggar aturan Sudoku), algoritma melanjutkan ke kotak berikutnya. Jika angka tersebut tidak valid, algoritma "mundur" (backtracking) dan mencoba angka lain.

Contoh:

```
main.py X
.vscode > main.py > ...
1 def solve_sudoku(grid):
2     """
3     Memecahkan teka-teki Sudoku menggunakan algoritma backtracking.
4
5     Args:
6     grid: List of lists representing the Sudoku grid.
7
8     Returns:
9     True if the Sudoku is solved, False otherwise.
10    """
11    for row in range(9):
12        for col in range(9):
13            if grid[row][col] == 0:
14                for num in range(1, 10):
15                    if is_valid(grid, row, col, num):
16                        grid[row][col] = num
17
18                        if solve_sudoku(grid):
19                            return True
20
21                        grid[row][col] = 0 # Backtrack
22
23    return False # No valid number found
24
25    return True # Sudoku is solved
26
27 def is_valid(grid, row, col, num):
28     """
29     Mengecek apakah angka 'num' valid untuk kotak (row, col).
30
31     Args:
32     grid: List of lists representing the Sudoku grid.
```



```

32     grid: List of lists representing the Sudoku grid.
33     row: Row index of the cell.
34     col: Column index of the cell.
35     num: Number to check.
36
37     Returns:
38     True if the number is valid, False otherwise.
39     """
40     # Periksa baris
41     for x in range(9):
42         if grid[row][x] == num:
43             return False
44
45     # Periksa kolom
46     for x in range(9):
47         if grid[x][col] == num:
48             return False
49
50     # Periksa kotak 3x3
51     start_row = row - row % 3
52     start_col = col - col % 3
53     for i in range(3):
54         for j in range(3):
55             if grid[i + start_row][j + start_col] == num:
56                 return False
57
58     return True
59
60 # Contoh penggunaan
61 grid = [
62     [5, 3, 0, 0, 7, 0, 0, 0, 0],
63     [6, 0, 0, 1, 9, 5, 0, 0, 0],
64     [0, 9, 8, 0, 0, 0, 0, 6, 0],
65     [8, 0, 0, 0, 6, 0, 0, 0, 3],
66     [4, 0, 0, 8, 0, 3, 0, 0, 1],
67     [7, 0, 0, 0, 2, 0, 0, 0, 6],
68     [0, 6, 0, 0, 0, 0, 2, 8, 0],
69     [0, 0, 0, 4, 1, 9, 0, 0, 5],
70     [0, 0, 0, 0, 8, 0, 0, 7, 9]
71 ]
72
73 if solve_sudoku(grid):
74     print("Solusi Sudoku:")
75     for row in grid:
76         print(row)
77 else:
78     print("Tidak ada solusi untuk Sudoku ini.")

```

Gambar 2.2 sudoku

Penjelasan:

1. **solve_sudoku(grid):** Fungsi ini menerima grid Sudoku sebagai input dan mencoba menyelesaikannya.
2. **for row in range(9) dan for col in range(9):** Loop ini iterasi melalui setiap kotak dalam grid.
3. **if grid[row][col] == 0:** Jika kotak kosong, algoritma mencoba mengisi kotak tersebut dengan angka yang valid.
4. **for num in range(1, 10):** Loop ini mencoba mengisi kotak dengan angka dari 1 hingga 9.
5. **if is_valid(grid, row, col, num):** Fungsi is_valid memeriksa apakah angka num valid untuk kotak tersebut.
6. **grid[row][col] = num:** Jika angka valid, algoritma mengisi kotak dengan angka tersebut.
7. **if solve_sudoku(grid):** Algoritma memanggil dirinya sendiri secara rekursif untuk mencoba menyelesaikan sisa grid.
8. **grid[row][col] = 0:** Jika angka tidak mengarah pada solusi yang valid, algoritma "mundur" dan mengembalikan kotak ke keadaan kosong.
9. **is_valid(grid, row, col, num):** Fungsi ini memeriksa apakah angka num valid untuk kotak (row, col) dengan memeriksa baris, kolom, dan kotak 3x3.

2.6 Algoritma Randomized

Algoritma randomized adalah algoritma yang menggunakan unsur acak dalam proses komputasinya untuk mencapai hasil yang diinginkan. Algoritma ini memperkenalkan keacakan untuk meningkatkan efisiensi atau menyederhanakan desain algoritma. Dengan memasukkan pilihan acak ke dalam prosesnya, algoritma randomized seringkali dapat memberikan solusi yang lebih cepat atau aproksimasi yang lebih baik dibandingkan dengan algoritma deterministik.

Jenis-Jenis Algoritma Randomized:

Algoritma randomized dapat diklasifikasikan menjadi dua jenis utama:

- **Algoritma Las Vegas:** Algoritma ini selalu menghasilkan solusi yang benar, tetapi waktu eksekusinya bervariasi secara acak. Contohnya adalah algoritma QuickSort, yang memiliki waktu eksekusi yang bervariasi tergantung pada urutan elemen input yang dipilih secara acak. Algoritma Las Vegas adalah jenis algoritma randomized yang menjamin menghasilkan solusi yang benar untuk suatu masalah, tetapi waktu eksekusinya bervariasi secara acak. Algoritma ini menggunakan unsur acak untuk mempercepat pencarian solusi, tetapi tidak mengorbankan keakuratan.

Konsep Dasar Algoritma Las Vegas:

- **Keacakan:** Algoritma Las Vegas menggunakan generator angka acak untuk membuat pilihan acak selama proses komputasi.
- **Solusi Tepat:** Algoritma ini selalu menghasilkan solusi yang benar untuk masalah yang diberikan.
- **Waktu Eksekusi Acak:** Waktu yang dibutuhkan untuk menyelesaikan masalah bervariasi tergantung pada pilihan acak yang dibuat.
- **Kecepatan:** Algoritma Las Vegas seringkali lebih cepat daripada algoritma deterministik untuk masalah yang kompleks, meskipun waktu eksekusinya tidak dapat diprediksi dengan pasti.

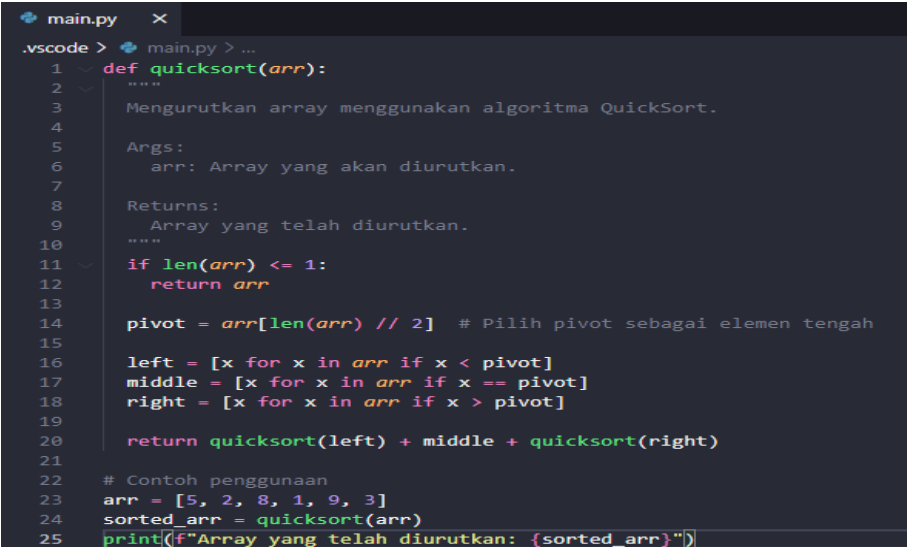
Contoh Penerapan Algoritma Las Vegas:

QuickSort: QuickSort adalah algoritma pengurutan yang menggunakan pendekatan "bagi dan taklukkan" (divide and conquer) untuk mengurutkan array. Ia bekerja dengan memilih sebuah elemen sebagai pivot, membagi array menjadi dua sub-array berdasarkan pivot, dan kemudian mengurutkan kedua sub-array secara rekursif.

Teori QuickSort:

- **Pemilihan Pivot:** QuickSort memilih sebuah elemen dari array sebagai pivot. Pivot ini digunakan untuk membagi array menjadi dua sub-array: sub-array yang berisi elemen yang lebih kecil dari pivot dan sub-array yang berisi elemen yang lebih besar dari pivot.
- **Pemisahan (Partition):** Array dipartisi berdasarkan pivot dengan memindahkan semua elemen yang lebih kecil dari pivot ke sebelah kiri pivot dan semua elemen yang lebih besar dari pivot ke sebelah kanan pivot.
- **Rekursi:** QuickSort kemudian secara rekursif mengurutkan kedua sub-array hingga seluruh array terurut.

Contoh:

A screenshot of a code editor window titled 'main.py'. The code defines a 'quicksort' function that sorts an array using the QuickSort algorithm. It includes docstrings for the function, its arguments, and its return value. The function uses a recursive approach, selecting a pivot, partitioning the array into left, middle, and right sub-arrays, and then combining them. A sample usage is provided at the bottom, creating an array [5, 2, 8, 1, 9, 3], sorting it, and printing the result.

```
1 def quicksort(arr):
2     """
3     Mengurutkan array menggunakan algoritma QuickSort.
4
5     Args:
6         arr: Array yang akan diurutkan.
7
8     Returns:
9         Array yang telah diurutkan.
10    """
11    if len(arr) <= 1:
12        return arr
13
14    pivot = arr[len(arr) // 2] # Pilih pivot sebagai elemen tengah
15
16    left = [x for x in arr if x < pivot]
17    middle = [x for x in arr if x == pivot]
18    right = [x for x in arr if x > pivot]
19
20    return quicksort(left) + middle + quicksort(right)
21
22 # Contoh penggunaan
23 arr = [5, 2, 8, 1, 9, 3]
24 sorted_arr = quicksort(arr)
25 print(f"Array yang telah diurutkan: {sorted_arr}")
```

Gambar 2.3 quick sort

Penjelasan:

1. **quicksort(arr):** Fungsi ini menerima array `arr` sebagai input dan mengurutkannya menggunakan QuickSort.
2. **if len(arr) <= 1:** Fungsi ini memeriksa apakah array memiliki panjang 1 atau kurang. Jika ya, array sudah terurut, dan fungsi mengembalikan array tersebut.
3. **pivot = arr[len(arr) // 2]:** Fungsi ini memilih elemen tengah dari array sebagai pivot.
4. **left = [x for x in arr if x < pivot]:** Fungsi ini membuat sub-array `left` yang berisi semua elemen yang lebih kecil dari pivot.
5. **middle = [x for x in arr if x == pivot]:** Fungsi ini membuat sub-array `middle` yang berisi semua elemen yang sama dengan pivot.
6. **right = [x for x in arr if x > pivot]:** Fungsi ini membuat sub-array `right` yang berisi semua elemen yang lebih besar dari pivot.
7. **return quicksort(left) + middle + quicksort(right):** Fungsi ini secara rekursif mengurutkan sub-array `left` dan `right` dan kemudian menggabungkannya dengan sub-array `middle` untuk menghasilkan array yang terurut.