# استفاده از الگوریتم‌های مختلف برای حل مسائل پیچیده

برای حل مسائل پیچیده، الگوریتم‌های مختلفی وجود دارند که بسته به نوع و ویژگی‌های مسئله، می‌توان از آن‌ها استفاده کرد. در اینجا، سه دسته مهم از الگوریتم‌ها شامل تقسیم و غلبه، برنامه‌نویسی پویا، و الگوریتم‌های گرافی معرفی و توضیح داده می‌شود.

---

## 1. الگوریتم‌های تقسیم و غلبه (Divide and Conquer)

الگوریتم‌های تقسیم و غلبه به روشی گفته می‌شود که مسئله به بخش‌های کوچکتر تقسیم می‌شود، سپس این بخش‌ها به‌صورت مستقل حل شده و در نهایت نتایج آنها با هم ترکیب می‌شود.

### مثال: مرتب‌سازی سریع (QuickSort)

مرتب‌سازی سریع یکی از الگوریتم‌های معروف تقسیم و غلبه است. این الگوریتم لیست را به دو بخش تقسیم کرده و هر بخش را به‌طور جداگانه مرتب می‌کند.

**پیچیدگی زمانی**: O(n log n) (در بهترین و متوسط حالت)

### کد پایتون برای QuickSort:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    less = [x for x in arr[1:] if x <= pivot]
    greater = [x for x in arr[1:] if x > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)

arr = [10, 7, 8, 9, 1, 5]
print(quicksort(arr))  # خروجی: [1, 5, 7, 8, 9, 10]
```

### مثال: مرتب‌سازی ادغامی (Merge Sort)

مرتب‌سازی ادغامی نیز یکی از الگوریتم‌های تقسیم و غلبه است که با تقسیم داده‌ها به دو بخش و ادغام آن‌ها پس از مرتب‌سازی، عمل مرتب‌سازی را انجام می‌دهد.

**پیچیدگی زمانی**: O(n log n)

### کد پایتون برای Merge Sort:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
```

```
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result

arr = [10, 7, 8, 9, 1, 5]
print(merge_sort(arr))  # خروجی: [10, 9, 8, 7, 5, 1]
```

---

## 2. برنامه‌نویسی پویا (Dynamic Programming)

برنامه‌نویسی پویا (DP) روشی است که برای حل مسائل بهینه‌سازی استفاده می‌شود، به‌ویژه برای مسائلی که دارای زیرمسائل تکراری هستند. این روش شامل ذخیره‌سازی نتایج محاسبات قبلی برای جلوگیری از محاسبه دوباره آن‌ها است.

### مثال: حل مسئله فیبوناچی با برنامه‌نویسی پویا

در اینجا به‌جای محاسبه مجدد هر عدد فیبوناچی، از نتایج محاسبات قبلی استفاده می‌شود.

**پیچیدگی زمانی**: O(n)

### کد پایتون برای فیبوناچی با DP:

```
def fibonacci(n):
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]

n = 10
print(fibonacci(n))  # خروجی: 55
```

### مثال: مسئله کوله‌پشتی (Knapsack Problem)

در این مسئله، هدف انتخاب اقلامی است که حداکثر ارزش را با در نظر گرفتن محدودیت وزن به دست آورند.

**پیچیدگی زمانی**: O(n * W) (n تعداد اقلام و W ظرفیت کوله‌پشتی)

### کد پایتون برای کوله‌پشتی:

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        for w in range(capacity + 1):
            if i == 0 or w == 0:
                dp[i][w] = 0
            elif weights[i - 1] <= w:
```

```
            dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
        else:
            dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]

weights = [1, 2, 3]
values = [10, 20, 30]
capacity = 5
print(knapsack(weights, values, capacity))  # خروجی: 50
```

---

## 3. الگوریتم‌های گرافی

الگوریتم‌های گرافی برای حل مسائل مختلف در گراف‌ها مثل مسیرهای کوتاه، بررسی اتصال گراف و یافتن مسیرها استفاده می‌شوند.

### الف. الگوریتم دیکسترا (Dijkstra’s Algorithm)

این الگوریتم برای یافتن کوتاه‌ترین مسیر از یک گره به دیگر گره‌ها در یک گراف وزن‌دار با وزن‌های غیرمنفی استفاده می‌شود.

**پیچیدگی زمانی**: O(E log V) (E تعداد یال‌ها و V تعداد گره‌ها)

### کد پایتون برای Dijkstra:

```
import heapq

def dijkstra(graph, start):
    pq = [(0, start)]
    distances = {start: 0}
    while pq:
        (dist, node) = heapq.heappop(pq)
        for neighbor, weight in graph[node]:
            old_cost = distances.get(neighbor, float('inf'))
            new_cost = dist + weight
            if new_cost < old_cost:
                distances[neighbor] = new_cost
                heapq.heappush(pq, (new_cost, neighbor))
    return distances

graph = {
    "A": [("B", 1), ("C", 4)],
    "B": [("C", 2), ("D", 5)],
    "C": [("D", 1)],
    "D": []
}

print(dijkstra(graph, "A"))  # خروجی: {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

### ب. الگوریتم فلوید-وارشال (Floyd-Warshall Algorithm)

این الگوریتم برای یافتن کوتاه‌ترین مسیر بین همه جفت گره‌ها در گراف‌های وزن‌دار استفاده می‌شود.

**پیچیدگی زمانی**: O(V^3) (V تعداد گره‌ها)

### کد پایتون برای Floyd-Warshall:

```
def floyd_warshall(graph):
    V = len(graph)
    dist = [[float('inf')] * V for _ in range(V)]
    for i in range(V):
        dist[i][i] = 0
    for u in range(V):
        for v, weight in graph[u]:
            dist[u][v] = weight

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

graph = [
    [(1, 3), (2, 8), (3, -4)],
    [(-1, 1), (2, 7)],
    [(3, 1)],
    [(0, 2), (1, 4)]
]

print(floyd_warshall(graph))
```

---

## نتیجه‌گیری

الگوریتم‌های تقسیم و غلبه، برنامه‌نویسی پویا و گرافی از تکنیک‌های مؤثر برای حل مسائل پیچیده هستند. با انتخاب الگوریتم مناسب برای هر مسئله، می‌توان عملکرد برنامه‌ها را به‌طور قابل‌توجهی بهبود داد.