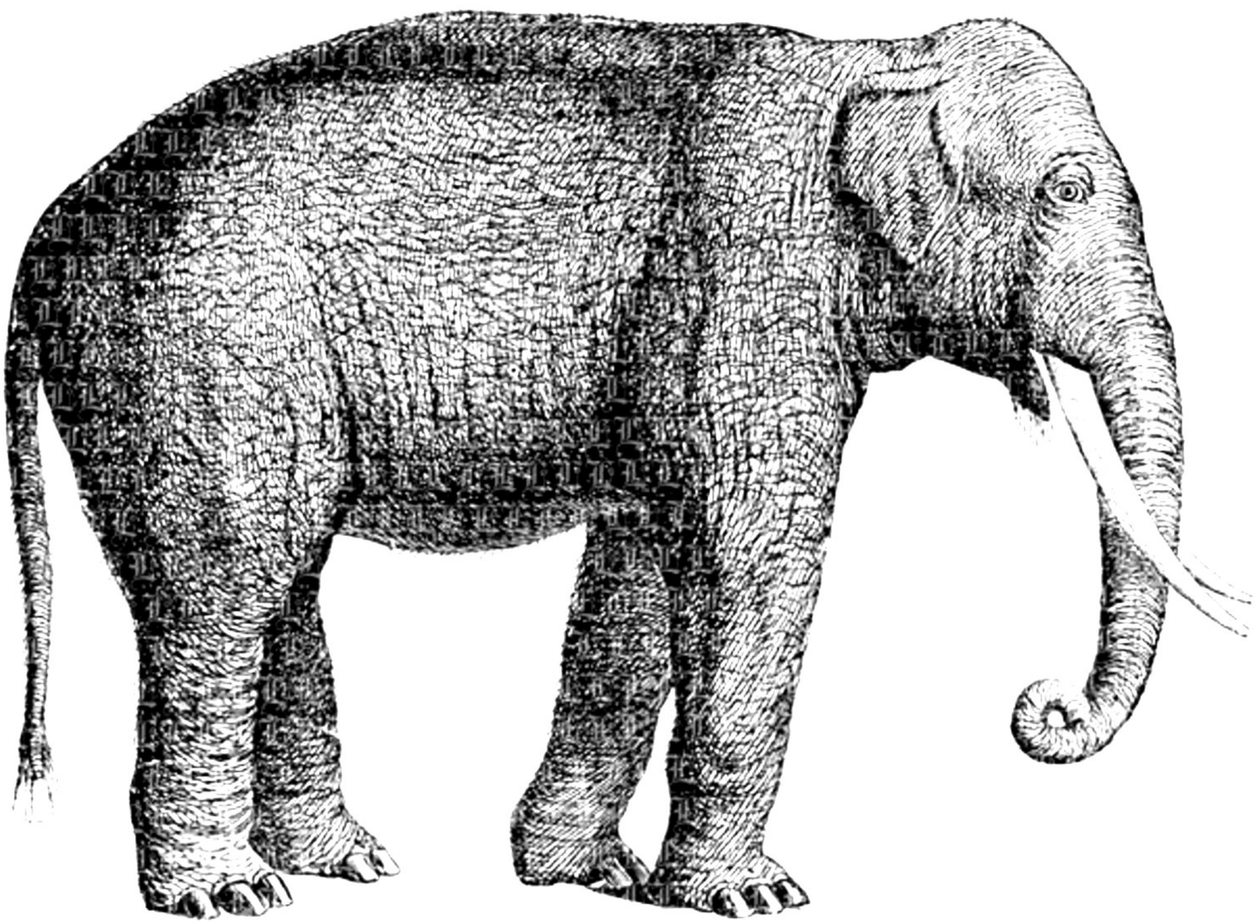
TAKING
INTRODUCTION TO ALGORITHMS

Production by TAKING



فهرست

مقدمه

الگوریتم‌ها قلب تپنده‌ی برنامه‌نویسی و مسابقات کدنویسی هستند. در این جزوه، شما با مهم‌ترین الگوریتم‌ها در دسته‌های مختلف آشنا خواهید شد، همراه با پیاده‌سازی آن‌ها در پایتون.

جست و جوی خطی

جست و جوی خطی یکی از ساده‌ترین روش‌های جستجو در آرایه‌ها و لیست‌ها است. در این روش، از ابتدای لیست شروع کرده و یکی‌یکی عناصر را بررسی می‌کنیم تا مقدار مورد نظر را پیدا کنیم یا به انتهای لیست برسیم.

مراحل اجرای جستجوی خطی :

1. **مقدار مورد نظر (کلید) را دریافت کن**
2. **از ابتدای لیست شروع کن**
3. **هر عنصر را با مقدار مورد نظر مقایسه کن**
4. **اگر مقدار یافت شد، ایندکس (شماره موقعیت) آن را برگردان**
5. **اگر مقدار در لیست نبود، مقدار 1- (یا هر مقدار دیگر برای نشان دادن عدم وجود) برگردان**

Edit Copy

```
def linear_search(arr, target):
    for i in range(len(arr)):  # پیمایش تمام عناصر لیست
        if arr[i] == target:  # بررسی مقدار فعلی با مقدار مورد نظر
            return i  # برگرداندن ایندکس مقدار پیدا شده
    return -1  # در صورتی که مقدار یافت نشد
```

جست و جوی باینری

جستجوی باینری یک الگوریتم کارآمد برای یافتن مقدار در یک لیست مرتب‌شده است. برخلاف جستجوی خطی، این روش از تقسیم و تسلط استفاده می‌کند و در هر مرحله، نیمی از عناصر را حذف می‌کند تا سریع‌تر مقدار مورد نظر را بیابد

1. مراحل اجرای جستجوی باینری :
2. ابتدا لیست باید مرتب باشد
3. مقدار وسط لیست را پیدا کن
4. مقدار وسط را با مقدار مورد نظر مقایسه کن :
5. اگر مقدار وسط همان مقدار مورد نظر است ← مقدار پیدا شده است
6. اگر مقدار وسط بزرگ‌تر از مقدار مورد نظر ← جستجو را در نیمه چپ ادامه بده
7. اگر مقدار وسط کوچک‌تر از مقدار مورد نظر ← جستجو را در نیمه راست ادامه بده
8. این فرایند را تکرار کن تا مقدار پیدا شود یا لیست خالی شود

روش بازگشتی (Recursive)

python

Edit Copy

```python
def binary_search(arr, target, left, right):
    if left > right:  # شرط توقف: اگر محدوده نامعتبر شد
        return -1

    mid = (left + right) // 2  # پیدا کردن مقدار وسط
    if arr[mid] == target:
        return mid  # مقدار پیدا شد!
    elif arr[mid] > target:
        return binary_search(arr, target, left, mid - 1)  # جستجو در نیمه چپ
    else:
        return binary_search(arr, target, mid + 1, right)  # جستجو در نیمه راست
```

روش حلقه (Iterative)

Edit Copy

```python
def binary_search_iterative(arr, target):
    left, right = 0, len(arr) - 1  # ابتدا و انتها

    while left <= right:
        mid = (left + right) // 2  # مقدار وسط

        if arr[mid] == target:
            return mid  # مقدار پیدا شد!
        elif arr[mid] > target:
            right = mid - 1  # جستجو در نیمه چپ
        else:
            left = mid + 1  # جستجو در نیمه راست

    return -1  # مقدار یافت نشد
```

مرتب سازی حبابی

مرتب‌سازی حبابی یکی از ساده‌ترین الگوریتم‌های مرتب‌سازی است که عناصر یک لیست را با مقایسه‌ی زوجی و جابه‌جایی مرتب می‌کند. در این روش، عناصر بزرگ‌تر مانند حباب به سمت انتهای لیست حرکت می‌کنند

1. **مراحل انجام مرتب‌سازی حبابی :**
2. **از ابتدای لیست شروع کن**
3. **هر دو عنصر مجاور را مقایسه کن :**
4. **اگر عنصر اول بزرگ‌تر از عنصر دوم بود، جایشان را عوض کن**
5. **اگر نبود، به جفت بعدی برو**
6. **این کار را برای کل لیست تکرار کن**
7. **در پایان هر مرحله، بزرگ‌ترین عدد به انتهای لیست منتقل می‌شود**
8. **این کار را ادامه بده تا کل لیست مرتب شود**

```python
def bubble_sort(arr):
    n = len(arr)  # اندازه لیست
    for i in range(n - 1):  # تعداد دفعات پیمایش
        swapped = False  # برای بهینه‌سازی (چک می‌کند که آیا جابه‌جایی انجام شد یا نه)
        for j in range(n - 1 - i):  # از ابتدا تا جایی که هنوز بررسی نشده
            if arr[j] > arr[j + 1]:  # اگر مقدار فعلی بزرگ‌تر از مقدار بعدی است
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # جابه‌جایی
                swapped = True
        if not swapped:  # اگر در یک دور هیچ جابه‌جایی انجام نشود، لیست مرتب است
            break
```

الگوریتم مرتب‌سازی انتخابی (Selection Sort)

مرتب‌سازی انتخابی یک الگوریتم ساده برای مرتب‌سازی است که در هر مرحله **کوچک‌ترین مقدار** را پیدا کرده و آن را در جای درستش قرار می‌دهد.

نحوه عملکرد مرتب‌سازی انتخابی

1. کل لیست را بررسی کن و **کوچک‌ترین مقدار** را پیدا کن.
2. آن را با **اولین عنصر** لیست جابه‌جا کن.
3. بخش مرتب‌شده را **یک خانه افزایش بده** (یعنی دیگر عنصر اول را بررسی نکن).
4. دوباره کوچک‌ترین مقدار را در بخش باقی‌مانده پیدا کن و در جای مناسبش قرار بده.
5. این کار را ادامه بده تا کل لیست مرتب شود.

پیاده‌سازی در پایتون

Edit Copy

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        min_index = i  # کوچک‌ترین مقدار است i فرض می‌کنیم عنصر
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:  # اگر مقدار کوچک‌تری پیدا شد
                min_index = j  # ایندکس کوچک‌ترین مقدار را به‌روز کن
        arr[i], arr[min_index] = arr[min_index], arr[i]  # مقدار کوچک‌تر با مقدار فعلی
```

الگوریتم مرتب‌سازی درجی (Insertion Sort)

مرتب‌سازی درجی یک روش ساده و کارآمد برای مرتب‌سازی **لیست‌های کوچک** است. این الگوریتم مشابه **بازی ورق** عمل می‌کند: کارت‌ها را یکی‌یکی برمی‌داریم و در جای مناسب خود در دست قرار می‌دهیم.

نحوه عملکرد مرتب‌سازی درجی

1. فرض می‌کنیم اولین عنصر لیست **از قبل مرتب شده است**.
2. از دومین عنصر شروع می‌کنیم و آن را در **بخش مرتب‌شده** در جای مناسب قرار می‌دهیم.
3. این کار را برای تمام عناصر تکرار می‌کنیم تا لیست کاملاً مرتب شود.

پیاده‌سازی در پایتون

Edit Copy

```
def insertion_sort(arr):
    for i in range(1, len(arr)):  # از دومین عنصر شروع می‌کنیم
        key = arr[i]  # مقدار فعلی که باید درج شود
        j = i - 1
        while j >= 0 and arr[j] > key:  # جابجایی عناصر بزرگ‌تر
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # قرار دادن مقدار در جای مناسب
```

الگوریتم مرتب‌سازی سریع (Quick Sort)

مرتب‌سازی سریع یکی از **بهترین و سریع‌ترین** الگوریتم‌های مرتب‌سازی است. این الگوریتم از روش **تقسیم و حل** (Divide and Conquer) استفاده می‌کند تا لیست را به بخش‌های کوچک‌تر تقسیم کرده و مرتب کند.

نحوه عملکرد مرتب‌سازی سریع

1. **یک مقدار محوری (Pivot) انتخاب کن** (معمولاً مقدار وسط، اولین، یا آخرین عنصر).
2. **لیست را به دو بخش تقسیم کن:**

- اعدادی که **کوچک‌تر یا مساوی** با مقدار محوری هستند.
- اعدادی که **بزرگ‌تر** از مقدار محوری هستند.

3. **به‌صورت بازگشتی** این مراحل را برای هر بخش انجام بده.
4. در نهایت، وقتی همه بخش‌ها مرتب شدند، لیست نهایی به‌دست می‌آید.

پیاده‌سازی در پایتون

Edit Copy

```
def quick_sort(arr):
    if len(arr) <= 1:  # شرط توقف (اگر لیست یک یا صفر عنصر داشت)
        return arr

    pivot = arr[len(arr) // 2]  # انتخاب مقدار محوری (وسط لیست)
    left = [x for x in arr if x < pivot]  # اعداد کوچکتر از محور
    middle = [x for x in arr if x == pivot]  # اعداد مساوی با محور
    right = [x for x in arr if x > pivot]  # اعداد بزرگتر از محور

    return quick_sort(left) + middle + quick_sort(right)  # ترکیب نتایج
```

الگوریتم مرتب‌سازی ادغامی (Merge Sort)

مرتب‌سازی ادغامی یک الگوریتم **کارآمد و پایدار** است که از روش **تقسیم و حل (Divide and Conquer)** استفاده می‌کند. این الگوریتم، لیست را به دو قسمت تقسیم کرده، هر قسمت را جداگانه مرتب می‌کند و سپس **دو قسمت مرتب‌شده را باهم ترکیب (ادغام) می‌کند.**

مراحل اجرای مرتب‌سازی ادغامی

1. **تقسیم:** لیست را **به دو نیمه مساوی تقسیم کن** تا زمانی که هر قسمت فقط **یک عنصر** داشته باشد.
2. **مرتب‌سازی بازگشتی:** هر نیمه را با استفاده از مرتب‌سازی ادغامی مرتب کن.
3. **ادغام:** دو نیمه مرتب‌شده را **به‌صورت یک لیست مرتب‌شده ترکیب کن.**

```python
def merge_sort(arr):
    if len(arr) <= 1:  # شرط توقف (اگر لیست یک یا صفر عنصر داشت)
        return arr

    mid = len(arr) // 2  # پیدا کردن وسط لیست
    left_half = merge_sort(arr[:mid])  # مرتب‌سازی نیمه چپ
    right_half = merge_sort(arr[mid:])  # مرتب‌سازی نیمه راست

    return merge(left_half, right_half)  # ادغام دو نیمه مرتب‌شده
```

```python
def merge(left, right):
    sorted_list = []
    i = j = 0

    # ادغام دو لیست مرتب‌شده
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1

    # اضافه کردن عناصر باقی‌مانده
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])

    return sorted_list
```

برنامه‌نویسی پویا (Dynamic Programming)

برنامه‌نویسی پویا یک تکنیک حل مسئله است که برای حل مسائل پیچیده‌ای که می‌توان آن‌ها را به زیرمسئله‌های کوچکتر تقسیم کرد، به‌کار می‌رود. این تکنیک معمولاً برای حل مسائل بهینه‌سازی (Optimization Problems) و شمارش (Counting Problems) استفاده می‌شود.

اصول برنامه‌نویسی پویا

برنامه‌نویسی پویا شامل دو اصل اصلی است:

1. **شکستن مسئله به زیرمسئله‌ها (Overlapping Subproblems):**
مسئله اصلی به چندین زیرمسئله کوچکتر تقسیم می‌شود. در برنامه‌نویسی پویا، این زیرمسئله‌ها معمولاً تکراری هستند و می‌توانند چندین بار حل شوند.
2. **ذخیره‌سازی نتایج زیرمسئله‌ها (Memoization or Tabulation):**
به جای محاسبه‌ی دوباره‌ی نتایج یک زیرمسئله، نتایج آن‌ها ذخیره می‌شوند تا در مواقع نیاز از آن‌ها استفاده شود. این عمل باعث کاهش زمان اجرای الگوریتم می‌شود.
برنامه‌نویسی پویا معمولاً به دو روش انجام می‌شود:

1. **ذخیره‌سازی (Memoization):**

- این روش به صورت **بازگشتی** است و نتایج زیرمسئله‌ها را در حافظه ذخیره می‌کند. وقتی یک زیرمسئله دوباره نیاز به محاسبه داشت، به جای محاسبه دوباره، از نتیجه‌ی ذخیره شده استفاده می‌شود.

2. **جدول‌بندی (Tabulation):**

- این روش به صورت **غیر بازگشتی** است و از یک جدول (یا آرایه) برای ذخیره‌سازی نتایج استفاده می‌کند. ابتدا زیرمسئله‌های کوچک‌تر حل می‌شوند و نتایج به تدریج برای مسائل بزرگ‌تر استفاده می‌شود.

ویژگی‌های برنامه‌نویسی پویا

- **کاهش پیچیدگی زمانی:**
با ذخیره‌سازی نتایج زیرمسئله‌ها، از انجام محاسبات تکراری جلوگیری می‌شود و در نتیجه پیچیدگی زمانی کاهش می‌یابد.
- **آسان برای پیاده‌سازی مسائل بهینه‌سازی:**
بسیاری از مسائل بهینه‌سازی و شمارش با استفاده از برنامه‌نویسی پویا به راحتی حل می‌شوند.
- **نیاز به فضای اضافی:**
برای ذخیره‌سازی نتایج زیرمسئله‌ها به فضای اضافی نیاز است. بسته به نوع الگوریتم، ممکن است فضای حافظه مورد نیاز زیاد باشد.

مسئله کوله‌پشتی (Knapsack Problem)

مسئله کوله‌پشتی یکی از معروف‌ترین مسائل در علم کامپیوتر و برنامه‌نویسی پویا است که در بسیاری از کاربردهای دنیای واقعی مانند مدیریت منابع، تخصیص بودجه و ... کاربرد دارد.

در این مسئله، شما تعدادی اشیاء دارید که هر کدام دارای وزن و ارزشی هستند. هدف این است که یک زیرمجموعه از این اشیاء را انتخاب کنید به‌طوری‌که مجموع وزن انتخاب شده از ظرفیت کوله‌پشتی تجاوز نکند و ارزش این انتخاب بیشینه شود.

شرح مسئله

- شما یک کوله‌پشتی با ظرفیت W دارید.
- تعدادی شیء وجود دارد که هر کدام وزن w_i و ارزش v_i دارند.
- هدف این است که یک زیرمجموعه از این اشیاء را انتخاب کنید به‌طوری‌که:
- مجموع وزن آن‌ها از ظرفیت کوله‌پشتی بیشتر نشود.
- مجموع ارزش آن‌ها بیشینه شود.

انواع مسئله کوله‌پشتی

1. **مسئله کوله‌پشتی ۰/۱ (Knapsack Problem 0/1):**

در این نسخه از مسئله، هر شیء فقط یک بار می‌تواند انتخاب شود (یعنی شما یا یک شیء را انتخاب می‌کنید یا انتخاب نمی‌کنید).

2. **مسئله کوله‌پشتی با تکرار (Unbounded Knapsack):**

در این نسخه، هر شیء می‌تواند چندین بار انتخاب شود.

ما در اینجا به بررسی **مسئله کوله‌پشتی ۰/۱** می‌پردازیم که بیشتر در برنامه‌نویسی پویا استفاده می‌شود.

فرمول‌بندی ریاضی

فرض کنید که n عدد شیء داریم که به‌طور کلی:

- w_i وزن شیء i-ام.
- v_i ارزش شیء i-ام.
- W ظرفیت کوله‌پشتی.

هدف این است که از بین اشیاء، زیرمجموعه‌ای انتخاب کنیم که:

1. مجموع وزن آن‌ها از W بیشتر نشود.
2. مجموع ارزش آن‌ها بیشینه باشد.

فرمول برنامه‌نویسی پویا

برای حل این مسئله با استفاده از برنامه‌نویسی پویا، یک آرایه dp تعریف می‌کنیم که:

- $dp[j]$ نشان‌دهنده بیشترین ارزشی است که می‌توان با ظرفیت j از کوله‌پشتی به‌دست آورد.

مراحل الگوریتم به صورت زیر است:

1. برای هر شیء i و برای هر ظرفیت j از ۰ تا W، بررسی می‌کنیم که آیا می‌توانیم این شیء را در کوله‌پشتی قرار دهیم.
2. اگر می‌توانیم، می‌بینیم که انتخاب این شیء باعث افزایش ارزش کلی می‌شود یا نه. در نهایت، $dp[W]$ نشان‌دهنده بیشترین ارزش قابل دستیابی با ظرفیت W خواهد بود.

مراحل الگوریتم

1. **مقداردهی اولیه:**
آرایه dp را به اندازه $W+1$ ایجاد می‌کنیم و تمام مقادیر آن را صفر قرار می‌دهیم.

2. **تکرار برای هر شیء:**
برای هر شیء، بررسی می‌کنیم که آیا با اضافه کردن آن به کوله‌پشتی، ارزش بیشتری به دست می‌آوریم یا نه. اگر بله، مقدار $dp[j]$ را به‌روزرسانی می‌کنیم.

3. **نتیجه نهایی:**
پس از بررسی تمام اشیاء، مقدار $dp[W]$ بیشترین ارزش ممکن را برای ظرفیت W خواهد داد.

```python
def knapsack(weights, values, W):
    n = len(weights)  # تعداد اشیاء
    dp = [0] * (W + 1)  # آرایه برای ذخیره بیشترین ارزش ممکن برای هر ظرفیت

    # بررسی تمام اشیاء #
    for i in range(n):
        # حرکت از ظرفیت بزرگ به کوچک تا از تکرار استفاده نکنیم #
        for j in range(W, weights[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

    return dp[W]
```

```python
# تست الگوریتم
weights = [2, 3, 4, 5]  # وزن اشیاء
values = [3, 4, 5, 6]   # ارزش اشیاء
W = 5  # ظرفیت کوله‌پشتی

result = knapsack(weights, values, W)
print(f"بیشترین ارزش ممکن: {result}")  # خروجی: 7
```

توضیح کد

1. ابتدا یک آرایه dp به اندازه $W+1$ ایجاد می‌کنیم که در آن $dp[j]$ بیشترین ارزش ممکن برای ظرفیت j را ذخیره می‌کند.
2. برای هر شیء i (که وزن و ارزش آن در آرایه‌های `weights` و `values` ذخیره شده‌اند)، از ظرفیت W به سمت صفر حرکت می‌کنیم و بررسی می‌کنیم که آیا می‌توانیم این شیء را در کوله‌پشتی قرار دهیم یا خیر.
3. برای هر ظرفیت j، بررسی می‌کنیم که آیا با قرار دادن این شیء، ارزش بیشتری به دست می‌آوریم یا نه.
4. در نهایت، مقدار $dp[W]$ بیشترین ارزش ممکن برای ظرفیت W خواهد بود.

فیبوناچی بهینه (Optimized Fibonacci)

در اینجا، ما قصد داریم بهینه‌ترین روش برای محاسبه دنباله فیبوناچی را بررسی کنیم. دنباله فیبوناچی به‌صورت زیر تعریف می‌شود:

$$F(0) = 0, \quad F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) \quad \text{برای } n > 1$$

روش‌های معمول برای محاسبه فیبوناچی

1. **روش بازگشتی ساده (Recursive):** این روش ساده‌ترین روش برای محاسبه فیبوناچی است که با استفاده از تعریف دنباله فیبوناچی کار می‌کند. اما این روش بسیار ناکارآمد است، زیرا در آن محاسبات تکراری زیادی انجام می‌شود.
2. **روش ذخیره‌سازی (Memoization):** این روش با استفاده از ذخیره‌سازی نتایج محاسبات قبلی، از محاسبات تکراری جلوگیری می‌کند. این روش به‌طور قابل توجهی سریع‌تر از روش بازگشتی ساده است.

3. **روش جدول‌بندی (Tabulation)**: در این روش، از یک آرایه برای ذخیره‌سازی نتایج فیبوناچی از پایین به بالا استفاده می‌شود. این روش نیز بسیار کارآمد است و زمان اجرای آن $O(n)$ است.
4. **روش بهینه (Space Optimized Fibonacci)**: این روش، بهینه‌ترین روش از لحاظ مصرف فضا است. در این روش به جای استفاده از آرایه‌ای برای ذخیره نتایج، فقط دو متغیر برای ذخیره آخرین دو مقدار نیاز است.

فیبوناچی بهینه (Space Optimized Fibonacci)

در این روش، به جای ذخیره‌سازی تمام مقادیر فیبوناچی، تنها دو مقدار آخر $F(n-1)$ و $F(n-2)$ ذخیره می‌شوند. این روش در مقایسه با روش‌های دیگر از نظر مصرف حافظه بسیار بهینه است.

الگوریتم فیبوناچی بهینه

Copy code

```python
def fibonacci_optimized(n):
    if n <= 1:
        return n

    prev2 = 0  # F(0)
    prev1 = 1  # F(1)

    for i in range(2, n + 1):
        current = prev1 + prev2  # F(n) = F(n-1) + F(n-2)
        prev2 = prev1  # بروزرسانی F(n-2)
        prev1 = current  # بروزرسانی F(n-1)

    return prev1  # F(n)

# تست
print(fibonacci_optimized(10))  # خروجی: 55
```

توضیحات کد

- ابتدا برای مقادیر $F(0)$ و $F(1)$ به صورت خاص مقداردهی می‌کنیم.
- سپس در حلقه‌ای از ۲ تا n، برای هر مقدار فیبوناچی، آخرین دو مقدار قبلی را به‌روزرسانی می‌کنیم.
- در نهایت، پس از اتمام حلقه، مقدار $F(n)$ در `prev1` ذخیره می‌شود و به‌عنوان نتیجه برمی‌گردد.

تحلیل زمانی و فضایی

- **پیچیدگی زمانی:**
این الگوریتم تنها یک حلقه از ۲ تا n دارد که پیچیدگی زمانی آن $O(n)$ است.
- **پیچیدگی فضایی:**
به دلیل استفاده از تنها دو متغیر برای ذخیره مقادیر قبلی، پیچیدگی فضایی الگوریتم $O(1)$ است.

نتیجه‌گیری

روش بهینه فیبوناچی (Space Optimized Fibonacci) یکی از بهترین روش‌ها از نظر مصرف حافظه است. این روش زمان اجرایی مشابه سایر روش‌ها دارد، اما به‌طور قابل توجهی حافظه کمتری مصرف می‌کند و برای مقادیر بزرگ n بسیار کارآمد است.

طولانی‌ترین زیررشته مشترک (Longest Common Substring)

مسئله **طولانی‌ترین زیررشته مشترک** به‌طور معمول در زمینه‌های پردازش رشته‌ها و مقایسه رشته‌ها مطرح می‌شود. در این مسئله، شما دو رشته دارید و هدف این است که طولانی‌ترین زیررشته‌ای را که در هر دو رشته وجود دارد پیدا کنید.

زیررشته (Substring):
زیررشته به مجموعه‌ای از کاراکترها گفته می‌شود که به‌طور متوالی از رشته‌ای گرفته شده‌اند. برای مثال، در رشته "abcdef" زیررشته‌های "abc" و "def" معتبر هستند.

شرح مسئله

- به شما دو رشته $S1$ و $S2$ داده شده است.
- هدف شما این است که طولانی‌ترین زیررشته‌ای را پیدا کنید که در هر دو رشته $S1$ و $S2$ مشترک باشد.

روش‌های مختلف حل مسئله

1. **روش brute-force (حمله به‌روش تمام احتمالات):** این روش شامل بررسی تمام زیررشته‌ها در هر دو رشته است و مقایسه کردن آن‌ها با هم. پیچیدگی زمانی این روش بسیار زیاد است و به $O(n^3)$ می‌رسد که در عمل برای رشته‌های بزرگ مناسب نیست.
2. **استفاده از جدول‌سازی (Dynamic Programming):** این روش با استفاده از برنامه‌نویسی پویا به‌طور مؤثرتر مسئله را حل می‌کند. به‌طور خاص، از یک جدول دو بعدی استفاده می‌شود که به هر دو رشته و زیررشته‌های مشترک آن‌ها اشاره دارد. پیچیدگی زمانی این روش $O(n \times m)$ است، که n و m طول دو رشته هستند.

الگوریتم با استفاده از برنامه‌نویسی پویا (Dynamic Programming)

در این روش از یک جدول دو بعدی استفاده می‌شود که هر عنصر $dp[i][j]$ نشان‌دهنده طول بزرگترین زیررشته مشترک از ابتدای رشته $S1$ تا i و ابتدای رشته $S2$ تا j است.

مراحل الگوریتم:

1. یک جدول dp به اندازه $(n+1) \times (m+1)$ ایجاد می‌کنیم.
   - $dp[i][j]$ نشان‌دهنده طول بزرگترین زیررشته مشترک از $S1[0 \dots i-1]$ و $S2[0 \dots j-1]$ است.
2. در صورت برابر بودن کاراکترهای $S1[i-1]$ و $S2[j-1]$، مقدار $dp[i][j]$ برابر با $dp[i-1][j-1]+1$ می‌شود.
3. در غیر این صورت، مقدار $dp[i][j]$ صفر می‌شود، زیرا نمی‌توان زیررشته مشترکی از آن‌ها ساخت.
4. در نهایت، بزرگ‌ترین مقدار در جدول dp طولانی‌ترین زیررشته مشترک خواهد بود.

```python
def longest_common_substring(S1, S2):
    n = len(S1)
    m = len(S2)

    # جدول برای ذخیره طول بزرگترین زیررشته مشترک
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    longest = 0  # طول بزرگترین زیررشته مشترک
    end_index = 0  # اندیس آخرین کاراکتر از بزرگترین زیررشته مشترک

    # پر کردن جدول dp
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if S1[i - 1] == S2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > longest:
                    longest = dp[i][j]
                    end_index = i - 1
            else:
                dp[i][j] = 0
```

```python
    # بازیابی طولانی‌ترین زیررشته مشترک
    lcs = S1[end_index - longest + 1:end_index + 1]

    return longest, lcs

# تست الگوریتم
S1 = "abcdefg"
S2 = "zcdemfgh"
length, substring = longest_common_substring(S1, S2)
print(f"طولانی‌ترین زیررشته مشترک: {substring} با طول {length}")
```

توضیحات کد

1. **تعریف تابع:**

تابع `longest_common_substring` دو رشته $S1$ و $S2$ را به‌عنوان ورودی می‌گیرد.

2. **جدول dp:**

یک جدول دو بعدی dp ایجاد می‌کنیم که بعد از اجرای الگوریتم، $dp[i][j]$ به طول بزرگ‌ترین زیررشته مشترک تا کاراکتر i از $S1$ و j از $S2$ اشاره خواهد کرد.

3. **پیمایش جدول:**

در هر تکرار حلقه‌ها، بررسی می‌کنیم که آیا کاراکترهای $S1[i-1]$ و $S2[j-1]$ برابر هستند یا نه. اگر برابر بودند، طول زیررشته مشترک را افزایش می‌دهیم.

4. **نتیجه نهایی:**

پس از پایان حلقه‌ها، بزرگ‌ترین مقدار در جدول dp که نشان‌دهنده طول بزرگ‌ترین زیررشته مشترک است، در متغیر `longest` ذخیره می‌شود. همچنین، زیررشته مشترک نیز بازیابی می‌شود و به‌عنوان نتیجه باز می‌گردد.

تحلیل زمانی و فضایی

- **پیچیدگی زمانی:**

الگوریتم نیاز به پیمایش تمام جدول dp دارد که در آن اندازه جدول $n \times m$ است (که n و m طول دو رشته هستند). بنابراین پیچیدگی زمانی این الگوریتم $O(n \times m)$ است.

- **پیچیدگی فضایی:**

به دلیل استفاده از یک جدول دو بعدی به اندازه $n \times m$، پیچیدگی فضایی این الگوریتم نیز $O(n \times m)$ است.

نتیجه‌گیری

این الگوریتم با استفاده از برنامه‌نویسی پویا به‌طور مؤثر و کارآمد طولانی‌ترین زیررشته مشترک دو رشته را پیدا می‌کند. این الگوریتم برای رشته‌های کوچک تا متوسط بسیار مناسب است.

مسئله تغییر سکه (Coin Change Problem)

مسئله تغییر سکه یکی از مسائل کلاسیک در برنامه‌نویسی پویاست که معمولاً برای بهینه‌سازی تعداد سکه‌ها یا کمترین تعداد سکه‌های مورد نیاز برای رسیدن به مبلغ مشخصی استفاده می‌شود.

تعریف مسئله

در این مسئله، شما تعدادی سکه با مقادیر مختلف دارید و باید برای مبلغ مشخصی، کمترین تعداد سکه‌هایی که جمع آن‌ها برابر با مبلغ مورد نظر شود، پیدا کنید. همچنین ممکن است شما بخواهید برای پیدا کردن تعداد مختلف راه‌ها برای ساخت مبلغ مورد نظر از سکه‌ها، راه‌حل‌هایی را پیدا کنید.

ورودی‌ها:

1. یک مجموعه از سکه‌ها با مقادیر مختلف.
2. مبلغ هدف که باید با استفاده از این سکه‌ها ساخته شود.

خروجی‌ها:

1. کمترین تعداد سکه‌ها برای ساخت مبلغ هدف.
2. اگر امکان ساخت مبلغ با استفاده از سکه‌ها وجود ندارد، خروجی باید «امکان‌پذیر نیست» یا مقداری مانند ∞ باشد.

الگوریتم برنامه‌نویسی پویا (Dynamic Programming) برای حل مسئله تغییر سکه

این مسئله را می‌توان با استفاده از برنامه‌نویسی پویا حل کرد. ما از یک جدول استفاده می‌کنیم که در آن هر عنصر نشان‌دهنده کمترین تعداد سکه‌ها برای ساخت مبلغ خاصی است.

مراحل الگوریتم:

1. **ایجاد یک آرایه:**
   یک آرایه `dp` با اندازه $M + 1$ (که M مبلغ هدف است) ایجاد می‌کنیم. مقدار `dp[i]` نشان‌دهنده کمترین تعداد سکه برای ساخت مبلغ i است.
2. **مقداردهی اولیه:**
   مقدار `dp[0]` برابر با 0 است، زیرا برای ساخت مبلغ صفر نیازی به سکه نداریم. دیگر مقادیر `dp[i]` ابتدا به ∞ مقداردهی می‌شوند، زیرا به‌طور فرضی هیچ راهی برای ساخت این مبالغ با سکه‌های فعلی وجود ندارد.
3. **محاسبه مقادیر:**
   برای هر سکه در مجموعه سکه‌ها، بررسی می‌کنیم که آیا می‌توانیم با استفاده از آن سکه، کمترین تعداد سکه‌ها برای ساخت مبلغ‌های مختلف را به‌روزرسانی کنیم.

4. **نتیجه:**
پس از پر کردن جدول `dp`، مقدار `dp[M]` نشان‌دهنده کمترین تعداد سکه‌ها برای ساخت مبلغ M است.

پیاده‌سازی در پایتون

Copy code

```
def coin_change(coins, amount):
    # ایجاد آرایه dp با اندازه (amount + 1)
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0  # برای مبلغ 0 نیاز به سکه نداریم

    # برای هر سکه
    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    # اگر مقدار dp[amount] همچنان بی‌نهایت باشد، یعنی ساختن این مبلغ ممکن نیست
    return dp[amount] if dp[amount] != float('inf') else -1

# تست
coins = [1, 2, 5]
amount = 11
result = coin_change(coins, amount)
print(f"کمترین تعداد سکه‌ها برای مبلغ {amount}: {result}")
```

توضیحات کد:

1. **آرایه `dp`**:
آرایه `dp` به‌طور اولیه با مقدار `inf` (بی‌نهایت) پر می‌شود تا نشان دهد که هیچ‌یک از مبالغ قابل دسترسی نیستند. تنها `dp[0]` برابر با صفر است، زیرا برای رسیدن به مبلغ صفر نیازی به سکه نداریم.
2. **بروزرسانی جدول:**
برای هر سکه در مجموعه سکه‌ها، از مقدار آن شروع می‌کنیم و بررسی می‌کنیم که برای ساخت هر مبلغ i آیا می‌توان از آن سکه استفاده کرد یا نه. به‌روزرسانی جدول به‌صورت $dp[i] = \min(dp[i], dp[i - \text{coin}] + 1)$ انجام می‌شود، که نشان‌دهنده کمترین تعداد سکه‌ها برای رسیدن به مبلغ i است.
3. **نتیجه:**
در پایان، اگر `dp[amount]` هنوز مقدار بی‌نهایت باشد، یعنی نمی‌توانیم مبلغ M را با استفاده از سکه‌های داده‌شده بسازیم و در این صورت -1 برمی‌گردانیم.

تحلیل زمانی و فضایی

- **پیچیدگی زمانی:**
زمان اجرای الگوریتم برابر با $O(n \times m)$ است، که در آن n تعداد سکه‌ها و m مبلغ هدف هستند. برای هر سکه، باید برای هر مبلغ از $coin$ تا $amount$ جدول را به‌روزرسانی کنیم.
- **پیچیدگی فضایی:**
فضای مصرفی این الگوریتم $O(m)$ است، زیرا تنها نیاز به آرایه‌ای با اندازه $amount + 1$ داریم.

نتیجه‌گیری

این الگوریتم با استفاده از برنامه‌نویسی پویا، به‌طور مؤثر و کارآمد کمترین تعداد سکه‌ها برای ساخت یک مبلغ هدف را محاسبه می‌کند. این الگوریتم زمان و فضای قابل قبولی دارد و برای مسائل مشابه بسیار مناسب است.

الگوریتم‌های بازگشتی (Recursive Algorithms)

الگوریتم‌های بازگشتی به الگوریتم‌هایی گفته می‌شود که برای حل یک مسئله، خود را برای حل زیرمسئله‌های کوچک‌تر فراخوانی می‌کنند. در واقع، یک الگوریتم بازگشتی معمولاً به صورت خودکار خودش را برای بخش‌های کوچکتر مسئله فراخوانی کرده و به حل آنها می‌پردازد تا در نهایت مسئله اصلی حل شود.

تعریف بازگشت در الگوریتم

بازگشت زمانی اتفاق می‌افتد که یک تابع یا الگوریتم، برای حل مسئله‌ای، خود را به‌طور مستقیم یا غیرمستقیم فراخوانی می‌کند. معمولاً برای اینکه الگوریتم به درستی کار کند، باید یک **شرط توقف** (Base Case) وجود داشته باشد تا از فراخوانی نامتناهی جلوگیری کند.

اجزای یک الگوریتم بازگشتی:

1. **پایه (Base Case):** شرایطی که به الگوریتم می‌گویند که دیگر نیازی به فراخوانی مجدد ندارد و به نتیجه نهایی می‌رسد.
2. **بازگشت (Recursive Case):** قسمت‌هایی از مسئله که به صورت بازگشتی به زیرمسئله‌های کوچک‌تر تقسیم می‌شود.

مثال‌ها:

1. محاسبه فاکتوریل یک عدد (Factorial):

یکی از ساده‌ترین مثال‌ها برای الگوریتم‌های بازگشتی، محاسبه فاکتوریل است. فاکتوریل یک عدد n به‌صورت $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$ تعریف می‌شود و رابطه بازگشتی آن به صورت زیر است:

$$n! = n \times (n-1)!$$

و

$$1! = 1$$

2. دنباله فیبوناچی (Fibonacci Sequence):

دنباله فیبوناچی به این صورت تعریف می‌شود:

$$F(0) = 0, \quad F(1) = 1$$

و

$$F(n) = F(n-1) + F(n-2) \quad \text{برای} \quad n \geq 2$$

3. جستجوی باینری (Binary Search):

الگوریتم جستجوی باینری برای جستجو در یک لیست مرتب‌شده استفاده می‌شود. اگر می‌خواهید به‌صورت بازگشتی به دنبال یک عنصر خاص در لیست بگردید، می‌توانید از الگوریتم جستجوی باینری استفاده کنید.

رابطه بازگشتی برای جستجو: اگر $\text{mid} = \frac{\text{low}+\text{high}}{2}$:

- اگر $\text{arr}[\text{mid}] == x$، آن‌وقت عنصر پیدا شده است.
- اگر $\text{arr}[\text{mid}] > x$، باید جستجو در نیمه چپ انجام شود.
- اگر $\text{arr}[\text{mid}] < x$، باید جستجو در نیمه راست انجام شود.

مزایا و معایب الگوریتم‌های بازگشتی

مزایا:

1. **سادگی پیاده‌سازی:** بسیاری از مسائل به‌ویژه مسائل تقسیم و غلبه، به‌راحتی با استفاده از الگوریتم‌های بازگشتی حل می‌شوند. این روش به‌ویژه در مسائلی مانند درخت‌ها و گراف‌ها کاربرد دارد.
2. **خوانایی کد:** کدهای بازگشتی می‌توانند خواناتر و ساده‌تر از کدهای غیر بازگشتی باشند.
3. **انعطاف‌پذیری:** بسیاری از مسائل مانند جستجو و مرتب‌سازی با استفاده از بازگشت قابل حل هستند.

معایب:

1. **هزینه حافظه:** الگوریتم‌های بازگشتی معمولاً فضای اضافی در حافظه مصرف می‌کنند. به‌ویژه در برخی موارد مانند محاسبه دنباله فیبوناچی، تابع‌های بازگشتی برای هر زیرمسئله، فضای جدیدی را در حافظه اختصاص می‌دهند.
2. **ممکن است منجر به فراخوانی بیش از حد شود:** اگر بازگشت به درستی طراحی نشود، ممکن است منجر به فراخوانی‌های تکراری زیاد و در نتیجه پیچیدگی زمانی زیاد شود.

نتیجه‌گیری

الگوریتم‌های بازگشتی روشی مؤثر برای حل مسائل پیچیده‌ای هستند که می‌توانند به زیرمسئله‌های کوچک‌تر تقسیم شوند. این الگوریتم‌ها معمولاً برای مسائل مانند جستجو، مرتب‌سازی، درخت‌ها و گراف‌ها مفید هستند. مهم است که در هنگام استفاده از الگوریتم‌های بازگشتی، به وجود یک **شرط توقف** توجه داشته باشیم تا از بروز فراخوانی‌های بی‌پایان جلوگیری شود.

مسئله برج‌های هانوی (Tower of Hanoi)

مسئله برج‌های هانوی یک مسئله کلاسیک در الگوریتم‌ها و برنامه‌نویسی است که به‌طور خاص برای معرفی مفاهیم بازگشتی و الگوریتم‌های بازگشتی استفاده می‌شود.

شرح مسئله:

در این مسئله، سه میله (یا برج) و تعدادی دیسک داریم که به ترتیب اندازه (از کوچک به بزرگ) روی یکی از میله‌ها قرار دارند. هدف این است که تمام دیسک‌ها را از میله اولیه به میله مقصد منتقل کنیم، با رعایت دو قاعده:

1. تنها یک دیسک می‌تواند در هر زمان جابجا شود.
2. در هر مرحله، دیسک‌ها باید به ترتیب اندازه جابجا شوند؛ یعنی دیسک بزرگتر هرگز روی دیسک کوچکتر قرار نگیرد.
3. از میله کمکی می‌توان استفاده کرد.

الگوریتم بازگشتی برای حل مسئله:

الگوریتم حل مسئله برج‌های هانوی به صورت بازگشتی انجام می‌شود:

1. **پایه (Base Case):** اگر تنها یک دیسک باقی مانده باشد، آن را به مقصد منتقل می‌کنیم.
2. **بازگشت (Recursive Case):** اگر بیشتر از یک دیسک باقی‌مانده باشد:

- ابتدا $n-1$ دیسک را به میله کمکی منتقل می‌کنیم (با استفاده از میله مقصد به‌عنوان میله کمکی).
- سپس دیسک آخر را (که بزرگترین دیسک است) به میله مقصد منتقل می‌کنیم.
- در نهایت، $n-1$ دیسک را از میله کمکی به میله مقصد منتقل می‌کنیم (با استفاده از میله اولیه به‌عنوان میله کمکی).

مراحل الگوریتم:

1. **انتقال دیسک‌های** $n-1$ از میله مبدأ به میله کمکی.
2. **انتقال دیسک** n (بزرگترین دیسک) از میله مبدأ به میله مقصد.
3. **انتقال دیسک‌های** $n-1$ از میله کمکی به میله مقصد.

```python
def hanoi(n, source, target, auxiliary):
    # شرط توقف (پایه)
    if n == 1:
        print(f"انتقال دیسک 1 از {source} به {target}")
        return

    # گام 1: انتقال n-1 دیسک از منبع به میله کمکی
    hanoi(n - 1, source, auxiliary, target)

    # گام 2: انتقال دیسک بزرگتر از منبع به مقصد
    print(f"انتقال دیسک {n} از {source} به {target}")

    # گام 3: انتقال n-1 دیسک از میله کمکی به مقصد
    hanoi(n - 1, auxiliary, target, source)

# تست الگوریتم
n = 3  # تعداد دیسک‌ها
hanoi(n, 'A', 'C', 'B')  # انتقال دیسک‌ها از میله A به میله C با استفاده از میله B
```

تحلیل زمانی و فضایی:

- **پیچیدگی زمانی:** الگوریتم برج‌های هانوی برای n دیسک، پیچیدگی زمانی $O(2^n)$ دارد، زیرا هر مرحله بازگشتی دوباره به دو زیرمسئله تقسیم می‌شود.
- **پیچیدگی فضایی:** پیچیدگی فضایی الگوریتم $O(n)$ است، زیرا در هر فراخوانی بازگشتی باید یک فضای جدید در پشته (Stack) ذخیره شود.

نتیجه‌گیری:

مسئله برج‌های هانوی یک مثال کلاسیک از الگوریتم‌های بازگشتی است که می‌تواند مفاهیم پایه‌ای برنامه‌نویسی و بازگشت را به‌خوبی نشان دهد. در این الگوریتم، مسئله به‌طور بازگشتی به زیرمسئله‌های کوچکتر تقسیم می‌شود و حل می‌شود تا در نهایت مسئله اصلی حل شود.

تولید جایگشت‌ها (Permutations)

جایگشت‌ها به ترتیب‌های مختلفی از یک مجموعه از عناصر گفته می‌شود که در آن‌ها ترتیب اهمیت دارد. برای مثال، جایگشت‌های مجموعه $[1, 2, 3]$ عبارتند از:

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

در اینجا، به ازای هر ترتیب از عناصر مجموعه، یک جایگشت جدید ایجاد می‌شود.

الگوریتم تولید جایگشت‌ها

یک روش ساده برای تولید تمام جایگشت‌ها از طریق **بازگشت (Recursion)** است. در این روش، ما یک عنصر را از مجموعه انتخاب کرده و مابقی مجموعه را به‌صورت بازگشتی برای تولید جایگشت‌های دیگر فراخوانی می‌کنیم.

مراحل الگوریتم تولید جایگشت‌ها:

1. اگر مجموعه فقط یک عنصر داشته باشد، تنها یک جایگشت وجود دارد که همان خود مجموعه است.
2. برای مجموعه‌ای که بیش از یک عنصر دارد:

- برای هر عنصر از مجموعه، آن را به‌عنوان اولین عنصر در جایگشت انتخاب کرده و بقیه عناصر را به‌صورت بازگشتی جایگشت می‌دهیم.
- این کار را برای تمام عناصر مجموعه تکرار می‌کنیم.

Copy code

```python
def generate_permutations(arr):
    # شرط توقف: اگر طول آرایه 1 یا کمتر باشد، تنها یک جایگشت وجود دارد
    if len(arr) == 1:
        return [arr]

    permutations = []
    for i in range(len(arr)):
        # انتخاب عنصر i از لیست
        current_element = arr[i]
        # باقی‌مانده لیست بدون عنصر i
        remaining_elements = arr[:i] + arr[i+1:]

        # بازگشت برای پیدا کردن جایگشت‌های باقی‌مانده
        for perm in generate_permutations(remaining_elements):
            # به ابتدای هر جایگشت حاصل i اضافه کردن عنصر
            permutations.append([current_element] + perm)

    return permutations

arr = [1, 2, 3]
result = generate_permutations(arr)
print("جایگشت‌ها:", result)
```

Copy code

```
جایگشت‌ها: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

توضیح کد:

1. **تابع** `generate_permutations` : این تابع بازگشتی است که برای هر عنصر در لیست، آن را به‌عنوان اولین عنصر در جایگشت قرار می‌دهد و سپس باقی‌مانده لیست را به‌صورت بازگشتی جایگشت می‌دهد.
2. **پایه بازگشت:** زمانی که لیست تنها یک عنصر داشته باشد، تنها یک جایگشت وجود دارد که همان خود عنصر است.
3. **مرحله بازگشت:** برای هر عنصر در لیست، آن را از لیست اصلی جدا می‌کنیم و سپس جایگشت‌های باقی‌مانده را محاسبه می‌کنیم.
4. **مجموعه جایگشت‌ها:** در نهایت، لیست تمام جایگشت‌ها برگشت داده می‌شود.

پیچیدگی زمانی:

پیچیدگی زمانی الگوریتم تولید جایگشت‌ها به‌طور کلی برابر است با $O(n!)$، زیرا تعداد جایگشت‌ها برای یک مجموعه از n عنصر، برابر با $n!$ است. این به این معنی است که برای هر عنصر در لیست، الگوریتم باید به‌صورت بازگشتی تمام جایگشت‌های ممکن را محاسبه کند.

نتیجه‌گیری:

- **تولید جایگشت‌ها** یک مسئله کلاسیک است که معمولاً با استفاده از الگوریتم‌های بازگشتی حل می‌شود.
- الگوریتم تولید جایگشت‌ها به‌طور معمول دارای پیچیدگی زمانی $O(n!)$ است.
- برای راحتی و بهینه‌سازی بیشتر، می‌توان از کتابخانه‌های استاندارد پایتون مانند `itertools` برای تولید جایگشت‌ها استفاده کرد.

الگوریتم‌های پس‌گرد (Backtracking)

الگوریتم پس‌گرد یک تکنیک برای حل مسائل پیچیده است که از راه‌حل‌های بالقوه استفاده می‌کند و سپس آن‌ها را در صورت عدم موفقیت، کنار می‌گذارد (برمی‌گردد) تا به راه‌حل‌های دیگر امتحان شده دست یابد. این الگوریتم به‌طور ویژه برای مسائلی که حل آن‌ها نیازمند جستجوی فضای بزرگی از حالت‌ها است، مناسب است.

در واقع، پس‌گرد یک استراتژی جستجو است که به دنبال راه‌حل‌های جزئی می‌گردد و اگر به نتیجه نرسید، به عقب برمی‌گردد و تلاش می‌کند تا از نقطه دیگری شروع کند.

مسئله چند وزیر (N-Queens Problem)

مسئله چند وزیر یک مسئله کلاسیک در الگوریتم‌ها است که در آن هدف این است که N وزیر را روی یک صفحه شطرنج $N \times N$ قرار دهیم به طوری که هیچ دو وزیری نتوانند یکدیگر را تهدید کنند. وزیری‌ها نمی‌توانند در یک ردیف، یک ستون یا یک قطر قرار گیرند.

این مسئله معمولاً با استفاده از الگوریتم‌های **بازگشتی** یا **پس‌گرد** (Backtracking) حل می‌شود.

تعریف مسئله:

- صفحه شطرنج یک جدول $N \times N$ است.
- باید N وزیر را روی صفحه قرار دهیم.
- هیچ دو وزیر نباید در یک ردیف، یک ستون یا یک قطر قرار بگیرند.
- هدف این است که تمام ترکیب‌های ممکن برای قرار دادن وزیری‌ها را پیدا کنیم که شرایط مسئله را رعایت کنند.

شرایط تهدید وزیری‌ها:

1. **ردیف:** هیچ دو وزیری نباید در یک ردیف قرار گیرند.
2. **ستون:** هیچ دو وزیری نباید در یک ستون قرار گیرند.
3. **قطر:** هیچ دو وزیری نباید در یک قطر قرار گیرند. (قطر اصلی از بالا چپ به پایین راست و قطر فرعی از بالا راست به پایین چپ)

روش حل: الگوریتم پس‌گرد (Backtracking)

1. **انتخاب ردیف‌ها:** شروع می‌کنیم به قرار دادن وزیرها در هر ردیف به ترتیب.
2. **بررسی امکان قرار دادن وزیر:** برای هر ردیف، بررسی می‌کنیم که آیا می‌توان وزیر را در ستون‌های مختلف قرار داد به طوری که تهدیدی ایجاد نشود.
3. **برگشت:** اگر در یک ردیف نتوانستیم وزیر را در هیچ ستون قرار دهیم، به عقب برگشته و تلاش می‌کنیم در ردیف قبلی تغییراتی اعمال کنیم.

مراحل الگوریتم پس‌گرد برای حل مسئله چند وزیر:

1. شروع از اولین ردیف و انتخاب ستون‌های مختلف برای قرار دادن وزیر.
2. برای هر انتخاب، بررسی می‌کنیم که آیا وزیر در آن ستون تهدید نمی‌کند (با توجه به ستون‌ها و قطرها).
3. اگر شرایط رعایت شد، وزیر را قرار می‌دهیم و به ردیف بعدی می‌رویم.
4. اگر به بن‌بست رسیدیم (نتوانستیم وزیر را در ردیف جاری قرار دهیم)، به عقب برگشته و ستون انتخابی قبلی را تغییر می‌دهیم.
5. زمانی که همه وزیرها در صفحه قرار گرفتند، راه‌حل پیدا شده است.

مثال:

برای $N = 4$، باید 4 وزیر را روی یک صفحه شطرنج 4×4 قرار دهیم که هیچ دو وزیر در یک ردیف، یک ستون یا یک قطر قرار نگیرند. یکی از راه‌حل‌ها ممکن است به شکل زیر باشد:

```
. Q . .
. . . Q
Q . . .
. . Q .
```

```python
def is_safe(board, row, col, N):
    # بررسی ستون
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve_n_queens(board, row, N):
    # اگر همه وزیرها قرار گرفتند
    if row == N:
        return True

    for col in range(N):
        # اگر جایگذاری وزیر در این خانه ایمن است #
        if is_safe(board, row, col, N):
            board[row] = col  # قرار دادن وزیر در این ستون
            # بازگشت برای ردیف بعدی
            if solve_n_queens(board, row + 1, N):
                return True
            # اگر نتوانستیم در این ردیف حل کنیم، برگشت میزنیم
            board[row] = -1

    return False  # اگر راه‌حل پیدا نشد
```

```python
def print_solution(board, N):
    for i in range(N):
        row = ['.'] * N
        row[board[i]] = 'Q'  # قرار دادن وزیر در جایگاه مناسب
        print(' '.join(row))

def n_queens(N):
    board = [-1] * N  # با همه خانه‌ها خالی NxN صفحه شطرنج
    if solve_n_queens(board, 0, N):
        print_solution(board, N)
    else:
        print("هیچ راه‌حلی پیدا نشد.")

# تست برای N=4
n_queens(4)
```

توضیح کد:

1. **تابع** `is_safe(board, row, col, N)`: این تابع بررسی می‌کند که آیا می‌توان یک وزیر در ردیف `row` و ستون `col` قرار داد یا خیر. برای این کار، بررسی می‌کند که وزیر در ردیف‌های قبلی در همان ستون یا همان قطر قرار نگرفته باشد.
2. **تابع** `solve_n_queens(board, row, N)`: این تابع بازگشتی است که برای هر ردیف، سعی می‌کند وزیر را در ستون‌های مختلف قرار دهد. اگر در ردیف جاری امکان قرار دادن وزیر وجود نداشته باشد، به عقب برمی‌گردد (پس‌گرد).
3. **تابع** `print_solution(board, N)`: پس از پیدا کردن راه‌حل، این تابع صفحه شطرنج را به‌صورت قابل فهم چاپ می‌کند. هر ستون حاوی موقعیت وزیر است که با علامت "Q" نشان داده شده است.
4. **تابع** `n_queens(N)`: این تابع تابع اصلی است که الگوریتم را اجرا می‌کند و صفحه شطرنج را بر اساس تعداد وزیرها `N` حل می‌کند.

پیچیدگی زمانی:

- پیچیدگی زمانی الگوریتم پس‌گرد برای حل مسئله N-Queens به‌طور کلی $O(N!)$ است. این به این دلیل است که برای هر ردیف، الگوریتم باید تمام ستون‌ها را بررسی کند و ممکن است در بدترین حالت به تمام ترکیب‌های ممکن نیاز داشته باشد.

نتیجه‌گیری:

- مسئله N-Queens یکی از مسائل کلاسیک در الگوریتم‌ها و علوم کامپیوتر است که به‌خوبی با استفاده از الگوریتم‌های پس‌گرد قابل حل است.
- این الگوریتم به‌ویژه برای مسائل ترکیبی که نیاز به جستجو در فضای بزرگ و بررسی ترکیب‌های مختلف دارند، بسیار مفید است.