Алгоритмын шинжилгээ ба зохиомж
Бие даалтын ажил 2
(F.CS301)

B210900007 Н.Тэмүүлэн

2024/11/12

Агуулга

1 Divide-and-Conquer

Divide-and-Conquer арга нь асуудлыг дэд асуудлуудад хуваан шийдвэрлэхэд чиглэгддэг. [1]

1.1 Үндсэн зарчим

Divide-and-Conquer арга нь асуудлыг дараах гурван үе шаттайгаар шийдвэрлэдэг:

1. **Divide**: Асуудлыг дэд асуудлууд болгон хуваах.
2. **Conquer**: Дэд асуудлуудыг тус бүрд нь шийдвэрлэх.
3. **Combine**: Шийдлүүдийг нэгтгэн анхны асуудлын хариуг гаргах.

Давуу талууд:

- Том массив дээр ажиллахад тохиромжтой, логарифмын түвшний хуваалт хийдэг тул хурдан. Рекурс ашигладаг тул ойлгоход хялбар, код бичихэд богино.

Сул талууд:

- Рекурсив дуудлага олон байвал санах ой их шаарддаг. Өргөтгөл хийхэд их хэмжээний давталт шаардлагатай байж магадгүй.

1.2 Жишээ бодлого

Массивын элементүүдийн нийлбэрийг олох

```
def sum_divide_and_conquer(arr, left, right):
    if left == right:
        return arr[left]
    mid = (left + right) // 2
    left_sum = sum_divide_and_conquer(arr, left, mid)
    right_sum = sum_divide_and_conquer(arr, mid + 1, right)
    return left_sum + right_sum

arr = [1, 3, 5, 7, 9]
print(sum_divide_and_conquer(arr, 0, len(arr) - 1))  # 25
```

2 Dynamic Programming

Dynamic Programming нь асуудлыг давхардсан тооцооллыг багасгах замаар шийдвэрлэдэг.

2.1 Гол санаа

1. Том асуудлыг жижиг дэд асуудлууд руу хуваах.
2. Дэд асуудлуудын шийдлүүдийг хадгалж, дахин ашиглах (**Memoization** буюу хадгалалт эсвэл **Tabulation** буюу хүснэгтчлэл).

Dynamic Programming-ийн давуу талууд

- Өмнөх үр дүнгүүдийг хадгалахын тулд дахин тооцоолох шаардлагагүй, ингэснээр цаг хугацаа хэмнэгдэнэ. DP нь зөв шийдлийг олох бөгөөд том асуудалд хүрэхэд ашигтай байдаг.

Dynamic Programming-ийн сул талууд

- Том массивуудыг хадгалж ашиглах шаардлагатай тул их хэмжээний санах ой шаардлагатай байж болно. Зарим үед DP алгоритмыг зөв гаргаж хэрэгжүүлэх нь төвөгтэй байж болно.

2.2 Жишээ бодлого

Массив дахь хамгийн их утгатай залгаа дэд массивын нийлбэрийг олох

```
def max_subarray_sum(arr):
    n = len(arr)
    max_ending_here = max_so_far = arr[0]

    for i in range(1, n):
        max_ending_here = max(arr[i], max_ending_here + arr[i])
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far

arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(arr))
```

3 Greedy Algorithms

Greedy алгоритм нь асуудлыг шийдвэрлэхдээ тухайн мөчид хамгийн их ашигтай шийдлийг сонгодог.

3.1 Greedy алгоритмын ажиллах үндсэн зарчим

Greedy алгоритм дараах үндсэн үе шатуудаас бүрддэг:

1. Сонголтын зарчим (Greedy Choice Property):
   - Тухайн үеийн хамгийн ашигтай шийдлийг сонгож, үүнийгээ оновчтой шийдлийн нэг хэсэг гэж үздэг.
   - Энэ алхам нь бүхэл асуудлыг оновчтой шийдэхэд хүргэх баталгаа байх ёстой.
2. Дэд бүтэцтэй харилцан хамаарал (Optimal Substructure):
   - Асуудлын оновчтой шийдлийг олохын тулд дэд асуудлуудыг шийдвэрлэх шаардлагатай.
   - Дэд асуудлуудын шийдэл нийлээд бүхэл асуудлын зөв шийдэл үүсгэх ёстой.

Давуу талууд:

- Алгоритмын цагийн нарийн төвөг багатай байж болно. Ихэвчлэн $O(n \log n)$ эсвэл $O(n)$ байх нь элбэг. Энгийн асуудлуудад амархан шийдэл өгдөг.

Сул талууд:

- Greedy алгоритм зөвхөн тухайн мөчид хамгийн сайн шийдлийг сонгодог тул үргэлж хамгийн оновчтой (optimal) шийдэлд хүрэхгүй. Зөвхөн **Greedy Choice Property** ба **Optimal Substructure** шинж чанартай асуудлуудад хэрэгжих боломжтой.

3.2 Жишээ бодлого

Интервалуудын хамгийн их тооны ажлыг сонгох

```
def interval_scheduling(intervals):
    intervals.sort(key=lambda x: x[1])
    count = 0
    last_end_time = 0

    for start, end in intervals:
        if start >= last_end_time:
            count += 1
            last_end_time = end

    return count

intervals = [(1, 3), (2, 5), (3, 9), (6, 8), (8, 10)]
print(interval_scheduling(intervals))
```

4 Харьцуулалт

4.1 Recursion vs Divide-and-Conquer

Жишээ бодлого: Хоёр тооны хамгийн их ерөнхий хуваагч (GCD) олох.

```
# Recursion
def gcd_recursive(a, b):
    if b == 0:
        return a
    return gcd_recursive(b, a % b)

# Divide-and-Conquer
def gcd_divide_and_conquer(a, b):
    while b != 0:
        if a > b:
            a -= b
        else:
            b -= a
    return a

a, b = 48, 18
print(gcd_recursive(a, b))          # 6
print(gcd_divide_and_conquer(a, b)) # 6
```

Үр дүн харьцуулалт: Recursion нь GCD-г олоход өгөгдсөн хоёр тоог үлдэгдлийн аргаар хуваах рекурсив аргыг ашигладаг бол Divide-and-Conquer нь хоёр тооноос томыг нь багасгах замаар илүү ойлгомжтой аргыг хэрэглэдэг.

Дүгнэлт: Аль аль нь GCD-г зөв олох боловч Recursion нь илүү хурдан, математик үндэслэлтэй.

4.2 Divide-and-Conquer vs Dynamic Programming

Жишээ бодлого: Массивын хамгийн урт өсөлттэй дараалалыг (Longest Increasing Subsequence) олох.

```
# Divide-and-Conquer
def lis_divide_and_conquer(arr):
    def helper(index, prev):
        if index == len(arr):
            return 0
        taken = 0
        if arr[index] > prev:
            taken = 1 + helper(index + 1, arr[index])
        not_taken = helper(index + 1, prev)
        return max(taken, not_taken)

    return helper(0, float('-inf'))
```

```python
# Dynamic Programming
def lis_dynamic_programming(arr):
    n = len(arr)
    dp = [1] * n
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

arr = [10, 22, 9, 33, 21, 50, 41, 60]
print(lis_divide_and_conquer(arr))          # 5
print(lis_dynamic_programming(arr))        # 5
```

Үр дүн харьцуулалт: Divide-and-Conquer нь бүх боломжийг рекурсив байдлаар шалгадаг тул том массив дээр цаг их шаарддаг. Dynamic Programming нь өмнөх үр дүнг хадгалан ашиглаж, хамаарал бүхий бодлогуудыг хурдан шийддэг.

Дүгнэлт: Longest Increasing Subsequence зэрэг бодлогуудад Dynamic Programming үр дүнтэй.

4.3 Dynamic Programming vs Greedy

Жишээ бодлого: Хамгийн их үр ашигтай ажиллах хугацааг сонгох (Activity Selection Problem).

```python
# Dynamic Programming
def activity_selection_dp(activities):
    n = len(activities)
    activities.sort(key=lambda x: x[1])
    dp = [0] * n
    dp[0] = 1

    for i in range(1, n):
        for j in range(i):
            if activities[j][1] <= activities[i][0]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)

# Greedy Algorithm
def activity_selection_greedy(activities):
    activities.sort(key=lambda x: x[1])
    count, last_end_time = 0, 0

    for start, end in activities:
        if start >= last_end_time:
            count += 1
            last_end_time = end
```

```
    return count

activities = [(1, 3), (2, 5), (0, 6), (5, 7), (8, 9), (5, 9)]
print(activity_selection_dp(activities))     # 4
print(activity_selection_greedy(activities)) # 4
```

Үр дүн харьцуулалт: Dynamic Programming нь бүх боломжит сонголтыг шалгадаг тул үр ашигтай хэдий ч илүү нарийн алгоритм шаарддаг. Greedy Algorithm нь тухайн үеийн хамгийн оновчтой сонголтыг хийх тул хурдан бөгөөд энгийн. [2]

Дүгнэлт: Activity Selection зэрэг асуудлуудад Greedy алгоритм хурдан бөгөөд үр дүнтэй.

5 Ашигласан эх сурвалжууд

Ашигласан ном

[1] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 4rd edition, 2022.

[2] Монгол бичгийн зөв бичлэгийн зөвлөмж. Зөв бичих зөвлөмж, 2024. URL https://zuv.bichig.dev.