ШИНЖЛЭХ УХААН ТЕХНОЛОГИЙН ИХ СУРГУУЛЬ

Мэдээлэл холбооны технологийн сургууль

Судалгааны материал

Хичээлийн нэр: Алгоритмын шинжилгээ ба зохиомж

(F.CS301)

(2024-2025 оны хичээлийн жил)

Судалгааны ажил гүйцэтгэсэн: **Э. Сүхбат/B221910061/**

Оршил

Алгоритмын загварчлал нь программчлалын үндсэн чухал хэсгүүдийн нэг бөгөөд олон төрлийн бодлого, асуудлыг үр ашигтай шийдвэрлэх арга замыг тодорхойлох боломжийг олгодог. Энэ тайланд бид алгоритмын гурван үндсэн аргачлал болох **Divide-and-Conquer**, **Dynamic Programming**, болон **Greedy Algorithms** сэдвүүдийг судалж, тэдгээрийн үндсэн зарчим, хэрэглээ, давуу болон сул талуудыг тодорхойлно.

Зорилго

Тайлангийн гол зорилго нь эдгээр алгоритмын аргачлалуудын онцлогийг харьцуулан судлрамчах, бодит жишээ бодлогоор тайлбарлах, мөн тэдгээрийг ямар нөхцөлд хамгийн үр дүнтэй ашиглаж болохыг тодорхойлох явдал юм.

Divide-and-Conquer (Хуваа-Засагла)

Divide-and-Conquer буюу **Хуваа-Засагла** алгоритм нь асуудлыг жижиг, бие даасан хэсгүүдэд хувааж, эдгээрийг тусад нь шийдвэрлээд эцэст нь нэгтгэх зарчимд үндэслэдэг.

Үндсэн алхмууд:

1. **Divide (Хуваах):** Анхны асуудлыг жижиг дэд асуудлуудад хуваана.
2. **Conquer (Засаглах):** Дэд асуудлуудыг тус бүрд нь шийднэ. Эдгээр нь ихэвчлэн рекурсив аргаар хийгддэг.
3. **Combine (Нэгтгэх):** Дэд асуудлуудын шийдлийг нэгтгэн анхны асуудлыг шийднэ.

Жишээ:

- **Merge Sort** – Өгөгдлийг эрэмбэлэхдээ массивыг хоёр хэсэгт хуваан, тус бүрийг эрэмбэлж, дараа нь нэгтгэнэ.
- **Quick Sort** – Гол цэг (pivot) сонгон өгөгдлийг хоёр хэсэгт хувааж, дараа нь хэсгүүдийг тусад нь эрэмбэлнэ.

Давуу тал:

- Рекурсив байдал нь зарим асуудлыг илүү ойлгомжтой шийдвэрлэхэд тусалдаг.
- Өндөр үр ашигтай шийдлүүдэд хүрэх боломжтой.

Сул тал:

- Рекурсийн санах ойн хэрэглээ ихэсдэг.

Жишээ:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2 left_half =
merge_sort(arr[:mid]) right_half =
```

```
merge_sort(arr[mid:]) return

merge(left_half, right_half) def

merge(left, right):

    sorted_array = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_array.append(left[i])
             i          += 1 else: sorted_array.append(right[j])
             j          += 1

    sorted_array.extend(left[i:])

    sorted_array.extend(right[j:])

    return sorted_array

array = [38, 27, 43, 3, 9, 82, 10]
sorted_array = merge_sort(array)
print("Sorted Array:", sorted_array)
```

Dynamic Programming (Динамик Программчлал)

Dynamic Programming буюу **Динамик Программчлал** алгоритм нь томоохон асуудлыг жижиг дэд асуудлуудад хувааж, тэдгээрийг нэг удаа тооцоолон хадгалах замаар асуудлыг шийдвэрлэдэг.

Үндсэн санаа:

1. Дэд асуудлууд давтагддаг тул өмнө нь шийдэгдсэн үр дүнг хадгалж дахин ашиглана (Memoization эсвэл Tabulation).
2. Том асуудлыг бага багаар шийдэж, үр дүнг ашиглан дараагийн алхмыг гүйцэтгэнэ.

Жишээ:

- **Fibonacci тооцоолол** – Өмнөх хоёр утгыг ашиглан дараагийн утгыг тооцоолно.
- **Knapsack Problem** – Тухайн багтаамжид хамгийн их ашиг олох зүйлсийг сонгоно.

Давуу тал:

- Давтагдсан тооцооллыг багасгаж, үр ашгийг нэмэгдүүлдэг.
- Том асуудлыг аажмаар шийдэхэд тохиромжтой.

Сул тал:

- Санах ойнч хэрэглээ ихэсдэг (таблиц эсвэл массив ашигладаг).
- Зарим асуудалд илүү энгийн алгоритмууд тохиромжтой байж болно.

Жишээ:

```
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1): for w in
        range(1, capacity + 1):
            if 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 = [2, 3, 4, 5] values = [3, 4, 5, 6] capacity = 5 print(f"Maximum value
in Knapsack: {knapsack(weights, values, capacity)}")
```

Greedy Algorithms (Шуналтай Алгоритм)

Greedy Algorithms буюу **Шуналтай Алгоритм** нь аливаа асуудлыг шийдэхдээ тухайн үеийн хамгийн ашигтай, хамгийн оновчтой алхмыг сонгож, эцсийн шийдэлд хүрэх аргачлал юм.

Үндсэн санаа:

1. Нэг алхамд хамгийн их үр ашигтай шийдлийг сонгоно.
2. Нийт шийдэлд хүрэхэд тухайн алхам тус бүр зөв байх ёстой.

Жишээ:

- **Dijkstra's Algorithm** – Хамгийн богино замыг олох.
- **Activity Selection Problem** – Хугацааны зөрчилгүй хамгийн олон үйл ажиллагааг сонгох.

Давуу тал:

- Хэрэгжүүлэхэд хялбар.
- Зарим асуудлыг богино хугацаанд шийдвэрлэдэг.

Сул тал:

- Зөвхөн тухайн үеийн ашигт анхаардаг тул бүхэл бүтэн асуудлын оновчтой шийдэлд хүрэхгүй байх магадлалтай.

Жишээ:

```
def fractional_knapsack(weights, values, capacity): ratios =
   [(values[i] / weights[i], i) for i in range(len(weights))]
   ratios.sort(reverse=True, key=lambda x: x[0])
   total_value = 0 for
   ratio, i in ratios:
      if weights[i] <= capacity:
         total_value += values[i]
         capacity -= weights[i]
      else:
         total_value += values[i] * (capacity /
   weights[i]) break return total_value

weights = [10, 20, 30] values = [60, 100, 120] capacity = 50 print(f"Maximum value in
Knapsack: {fractional_knapsack(weights, values, capacity)}")
```

Recursion vs Divide-and-Conquer

```
def factorial(n):
   if n == 0:
      return 1
   return n * factorial(n - 1)

print(factorial(5)) # Output: 120
```

- Рекурси нь ерөнхий арга бөгөөд асуудлыг тодорхой хувааж шийддэггүй бол хувааж-явуулах нь асуудлыг хувааж, хоорондоо давхацдаггүй дэд асуудлуудаар шийддэг.

Divide-and-Conquer vs Dynamic Programming

```
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]

print(fibonacci(5)) # Output: 5
```

- Хувааж-явуулах нь дэд асуудлуудыг хараат бус байдлаар шийддэг бол динамик программчлал нь давхацсан дэд асуудлуудыг хадгалж, дахин хэрэглэдэг.

Dynamic Programming vs Greedy

```
def knapsack(weights, values, capacity):
   n = len(weights)
```

```
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1): for w in
        range(1, capacity + 1):
            if 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 = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5
print(knapsack(weights, values, capacity)) # Output: 7
```

- Динамик программчлал нь бүх боломжит шийдлүүдийг шалгаж, хамгийн сайн шийдлийг олдог бол гриди нь зөвхөн хамгийн сайн шийдлийг сонгон авч, бүх тохиолдолд хамгийн сайн шийдлийг олж чадахгүй.

Дүгнэлт:

- **Рекурси vs Хувааж-Явуулах**: Рекурси нь ерөнхий арга бөгөөд функц өөрийгөө дууддаг байдлаар асуудлыг шийддэг. Харин хувааж-явуулах нь тодорхой бүтэцтэй, асуудлыг хувааж, давхацдаггүй дэд асуудлуудыг шийдэж, тэдгээрийг нэгтгэн шийдлийг олдог. Хувааж-явуулах нь илүү нарийвчилсан бөгөөд үр дүнтэй шийдэлд хүргэдэг.
- **Хувааж-Явуулах vs Динамик Программчлал**: Хувааж-явуулах нь асуудлыг хараат бус дэд хэсгүүдэд хувааж шийддэг бол динамик программчлал нь давхацаж буй дэд асуудлуудыг шийдэж, үр дүнг нь хадгалан дахин тооцоолохгүйгээр үр дүнд хүргэдэг. Динамик программчлал нь илүү оновчтой, үр ашигтай, дахин тооцоолохоос сэргийлдэг.
- **Динамик Программчлал vs Гриди**: Динамик программчлал нь бүх боломжит шийдлүүдийг шалгаж, хамгийн сайн шийдлийг олж гаргадаг бөгөөд үүний үр дүнд оновчтой шийдлийг баталгаажуулдаг. Харин гриди нь зөвхөн тухайн үеийн хамгийн сайн сонголтыг хийж, бүх тохиолдолд хамгийн сайн шийдлийг олж чадахгүй байж болно. Грийн нь хурдан бөгөөд энгийн боловч зарим тохиолдолд төгс шийдлийг олж чадахгүй.

Энэ бүх алгоритмууд нь өөрийн давуу болон сул талуудтай бөгөөд асуудал бүрийн онцлогт тохирсон хамгийн тохиромжтой аргыг сонгох нь чухал юм.