ШИНЖЛЭХ УХААН ТЕХНОЛОГИЙН ИХ СУРГУУЛЬ
Мэдээлэл, холбооны технологийн сургууль

БИЕ ДААЛТ 2-ЫН АЖЛЫН ТАЙЛАН

Алгоритмын шинжилгээ ба зохиомж

2024-2025 оны хичээлийн жилийн намар

Шалгасан багш: **Д.Батмөнх**

Бие даалтын ажил гүйцэтгэсэн: **Б.Насанжаргал**

2024

Энэхүү бие даалтын ажлын хүрээнд оюутан дараах сэдвүүдийг судалж, өөрийн ойлгосноо жишээ бодлогоор тайлбарлан бичнэ

1. ***Divide-and-Conquer (Merge Sort with Divide and Conquer)***

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

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

    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged

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

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

    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    return merge(left_sorted, right_sorted)

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

Example output is :
Sorted array: [3, 9, 10, 27, 38, 43, 82]

Тайлбар:
Divide and Conquer аргыг ашиглан Merge sort оор array доторх өгөгдлийг sort хийсэн байгаа. Divide and Conquer method нь алгоритм боловсруулах аргуудын нэг бөгөөд асуудлыг жижиг хэсгүүдэд хувааж , эдгээр хэсгүүдийг тус тусад нь шийдэж , дараа нь нэгтгэн нийт асуудлыг шийдвэрлэх аргачлал юм.

Жишээ бодлогын хувьд тайлбарлавал:

Merge Function (Нэгтгэх функц)

Энэ функц нь хоёр эрэмбэлэгдсэн жагсаалтыг нэгтгэнэ.

- merged:Хоёр жагсаалт (left, right)-ыг нэгтгэх жагсаалт.
- i, j: Хоёр жагсаалтын индексүүд.
- while давталт нь аль жагсаалтын элемент бага байгааг шалгаж, хамгийн бага элементийг merged жагсаалтад нэмнэ.
- Нэг жагсаалтын бүх элемент нэмэгдсэний дараа үлдсэн элементүүдийг нэмэхийн тулд extend() ашиглана.
- Ингээд хоёр жагсаалтыг нэгтгээд merged жагсаалтыг буцаана.

Merge Sort Function (Merge Sort функц)

Энэ функц нь массиваас ялгаатайгаар merge sort алгоритмыг ашиглан эрэмбэлдэг.

- if len(arr) <= 1: return arr: Массивын урт 1 буюу түүнээс бага байвал, уг массиваа шууд буцаана.
- mid = len(arr) // 2: Массивыг дундуур нь хувааж mid индексийг олно.
- left_half = arr[:mid]: Массивын зүүн хэсгийг left_half-д хадгална.
- right_half = arr[mid:]: Массивын баруун хэсгийг right_half-д хадгална.
- left_sorted = merge_sort(left_half): Зүүн хагасыг дахин merge_sort ашиглаж эрэмбэлнэ.
- right_sorted = merge_sort(right_half): Баруун хагасыг дахин merge_sort ашиглаж эрэмбэлнэ.
- return merge(left_sorted, right_sorted): Зүүн болон баруун эрэмбэлэгдсэн хэсгүүдийг нэгтгэж буцаана

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

Энэ нь merge_sort функцийг ашиглан массиваа эрэмбэлж , үр дүнг хэвлэнэ

Үр дүнд :
Sorted array: [3,9,10,27,38,43,82]

2. ***Dynamic Programming (Find Fibonacci numbers)***

Динамик программчлал нь том асуудлыг жижиг асуудлуудад хувааж шийдвэрлэх аргачлал юм. Энэ нь ихэвчлэн давталтын асуудлыг үр дүнтэйгээр шийдвэрлэхэд ашиглагддаг. Нэг жишээ болох Fibonacci тоонуудыг олох явц дээр тайлбарлая.

```
def fibonacci(n)
        # herev n too n 0 esvel 1 bol shuud butsaana
        if n <= 1:
          return n

        # fib massiviig uusgene, ehnii 2 utgiig 0 ,1 gej togtoono
        fib = [0] * ( n+1)
        fib[0] = 0
        fib[1] = 1

        # Dinamic programchlaliin argaar fibonacci toonuudig oly
        for i in range( 2, n+1 )
          fib[i] = fib[i-1] + fib[i-2]

        return fib[n]

# jishee hereglee:
n=10
print(f"fibonacci ({n}) = { fibonacci (n)}")
```

Тайлбар:

1. Функцийн тодорхойлолт: fibonacci(n) функц нь n дахь fibonacci тоог олоход ашиглагдана.
2. Суурь нөхцөл: Хэрэв n нь 0 эсвэл 1 бол шууд буцаана.
3. fib массив: fib массивыг үүсгэн, эхний хоёр утгыг 0 болон 1 гэж тогтооно.
4. Давталт: 2-оос эхлэн n хүртэлх бүх тоонуудын fibonacci утгуудыг олохын тулд давталт хийнэ. Динамик программчлалыг ашиглан өмнөх хоёр утгыг нэгтгэн одоогийн утгыг олох.
5. Буцаах: Эцэст нь, fib[n]-ийг буцаана.

Жишээ үр дүн :

```
Fibonacci(10) = 55
```

Динамик программчлалын аргачлал нь давталтын асуудлыг санах ой ашиглан шийдвэрлэж , цаг хугацааны хувьд илүү үр дүнтэй болгодог

3. *Greedy Algorithms (Knapsack Problem)*

(Greedy) алгоритм нь асуудлыг шийдвэрлэхдээ хамгийн их ашиг олох алхмыг сонгож шийдвэрлэх арга юм. Энэ аргачлалын нэг жишээ нь Хувийн үүргэвч (Knapsack) асуудал юм.

```
# Объектын бүтэц
class Item:
        def __init__(self, value, weight):
                self.value = value
                self.weight = weight
# Greedy алгоритм
def fractional_knapsack(items, capacity):
        items.sort(key=lambda item: item.value/item.weight, reverse=True)

        total_value = 0
        for item in items:
                if capacity >= item.weight:
                        capacity -= item.weight
                        total_value += item.value
                else:
                         total_value += item.value * (capacity / item.weight)
                        break
                return total_value
        # Жишээ өгөгдөл
        items = [Item(60, 10), Item(100, 20), Item(120, 30)]
        capacity = 50

        # Үүргэвчний асуудлыг шийдвэрлэх
        max_value = fractional_knapsack(items, capacity)
        print(f"Хамгийн их ашиг: {max_value}"
```

Тайлбар:

1. Item класс: Item класс нь эд зүйлийн үнэ (value) болон жинг (weight) тодорхойлдог.
2. fractional_knapsack функц:
    - Item ийн үнэлгээгээр (value/weight) ангилж эрэмбэлнэ.
    - total_value хувьсагчийг нийт ашиг олоход ашиглана.
    - Хэрэв Item ний жин үүргэвчний хязгаарт багтаж байвал, жинг хасаж, ашгийг нийтэд нэмнэ.
    - Хэрэв Item ний жин үүргэвчний хязгаарт багтахгүй бол, хуваах аргаар ашгийг олоод циклээс гарна.
3. Жишээ өгөгдөл: Хэд хэдэн Item ыг жингийн хязгаарт багтаах.
4. Үр дүн: Хамгийн их ашгийг хэвлэнэ.

Үр дүн: Хамгийн их ашиг: 240.0

50 жингийн хязгаартай үүргэвчний хамгийн их ашиг нь 240 болно. Энэ алгоритм нь "Greedy" аргыг ашигласан тул өгөгдсөн жингийн хязгаарт хамгийн их ашиг олох эд зүйлийг сонгож, жижиг асуудлыг хурдан шийдэж байгааг харуулна.

Улмаар дараах харьцуулалтуудыг хийж, мөн жишээгээр тайлбарлан бичнэ:

1. Recursion vs Divide-and-Conquer

Жишээ: Рекурсив функцээр факториалыг тооцоолох

Факториал гэдэг нь өгөгдсөн тооны бүх тоог үржүүлсэн утга юм.
Код:

```
def factorial_recursive(n):
    if n <= 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)
n = 5
print(f"{n}-н факториал нь:",factorial_recursive(n))
```

Тайлбар:

1. factorial_recursive функц:
    - Хэрэв n нь 1 буюу түүнээс бага байвал, 1-ийг буцаана.
    - Бусад тохиолдолд, n болон n-1-ийн факториалыг үржүүлж буцаана.
2. Жишээ:
    - n = 5 байхад, factorial_recursive(5) нь 5 * 4 * 3 * 2 * 1 = 120 болно.

Recursion: Хувьсамжтай асуудлуудад ашиглагддаг бөгөөд давталтын аргаар асуудлыг шийдвэрлэдэг. Энд гол түлхүүр нь функц өөрийгөө дуудаж асуудлыг шат дараалан шийдвэрлэх явдал юм.

Жишээ: Find Max and Min using Divide and Conquer

Код:

```
def find_min_and_max(arr, low, high):
    if low == high:  # If there's only one element
        return (arr[low], arr[high])

    if high == low + 1:  # If there are only two elements
        if arr[low] > arr[high]:
            return (arr[high], arr[low])
        else:
            return (arr[low], arr[high])

    mid = (low + high) // 2

    min1, max1 = find_min_and_max(arr, low, mid)
    min2, max2 = find_min_and_max(arr, mid+1, high)
```

```
        overall_min = min(min1, min2)
        overall_max = max(max1, max2)

         return (overall_min, overall_max)

# Жишээ хэрэглээ
arr = [12, 5, 8, 19, 22, 7, 15]
n = len(arr)
min_val, max_val = find_min_and_max(arr, 0, n-1)
print(f"Хамгийн бага утга: {min_val}, Хамгийн их утга: {max_val}")
```

Тайлбар:

1. find_min_and_max функц:
    - low == high: Хэрэв массив нэг элементтэй байвал тэр элементийг хамгийн бага болон их утга гэж тодорхойлно.
    - high == low + 1: Хэрэв массив хоёр элементтэй байвал тэдгээрийн хооронд хамгийн бага болон их утгыг тодорхойлно.
    - mid = (low + high) // 2: Массивыг дундуур нь хувааж, хоёр хэсэгт хуваана.
    - Зүүн болон баруун хэсгүүдийн хамгийн бага болон их утгыг тус туст нь олно.
    - Бүхэл массивын хамгийн бага болон их утгыг олно.
2. Жишээ:
    - Массивыг дундуур нь хувааж, зүүн болон баруун хэсгүүдийн хамгийн бага болон их утгыг олно.
    - arr = [12, 5, 8, 19, 22, 7, 15] байхад, хамгийн бага утга нь 5, хамгийн их утга нь 22 болно.

Divide and Conquer: Энэ бол "Divide and Conquer" аргачлалын багахан жишээ бөгөөд том асуудлыг жижиг хэсгүүдэд хувааж, эдгээр хэсгүүдийг тус тусад нь шийдвэрлээд нэгтгэн нийт асуудлыг шийдвэрлэдэг. Энэ аргыг ихэвчлэн эрэмбэлэх болон хайх алгоритмуудад ашигладаг.

Ялгаатай байдлын хураангуй:
Хамрах хүрээ: Рекурс нь асуудлын хэмжээг алхам алхмаар багасгахад чиглэдэг. "Divide and Conquer" нь асуудлыг бие даасан дэд асуудал болгон хувааж, тэдгээрийн шийдлүүдийг нэгтгэдэг.

Overhead: Recursion нь дуудлагын стекийн ачаалал ихтэй байж болно. Divide and Conquer нь санах ойг илүү үр дүнтэй болгох боломжтой.

2. Divide-and-Conquer vs Dynamic Programming

1. Фибоначчигийн тоонуудыг ол using (Divide and Conquer)

Код:

```
def fib_divide_and_conquer(n):
    if n <= 1:
        return n
    return fib_divide_and_conquer(n-1) + fib_divide_and_conquer(n-2)
# Жишээ хэрэглээ:
n = 10
print(f"Fibonacci({n}) = {fib_divide_and_conquer(n)}")
```

Тайлбар:

- **Divide:** Фибоначчигийн тоог олохын тулд n-1 болон n-2 тоонуудыг олох.
- **Conquer:** n-1 болон n-2 тоонуудыг олохын тулд дахин хувааж, тухайн тоонуудыг олох.
- **Нэгтгэх:** n-1 болон n-2 тоонуудыг нэмэгдэж, нийт Фибоначчигийн тоог олох.

2. Фибоначчигийн тоонуудыг ол using (Dynamic Programming)

Код:

```
def fib_dynamic_programming(n):
    fib = [0, 1]
    for i in range(2, n + 1):
        fib.append(fib[i-1] + fib[i-2])
    return fib[n]
# Жишээ хэрэглээ:
n = 10
print(f"Fibonacci({n}) = {fib_dynamic_programming(n)}")
```

Тайлбар:

- **Memoization:** Өмнө олоод хадгалсан Фибоначчигийн утгуудыг хадгалж, дахин тооцоолоход ашиглана.
- **Илүү үр ашигтай:** Тооцоолол нэгэнт хийгдсэн утгуудыг дахин ашиглах тул цаг хугацааны хувьд илүү хурдан.
- **Санах ой ашиглалт:** Массив ашиглаж утгуудыг хадгалах тул санах ой ашиглалт өндөр байж болно.

Ялгаа:

1.Давталтын зардал:

- Divide and Conquer: Дахин дахин давталт хийж утгуудыг олно, үүний улмаас цаг хугацааны хувьд зардал өндөр.
- Динамик программчлал: Өмнө олж хадгалсан утгуудыг дахин ашиглах тул цаг хугацааны хувьд илүү үр ашигтай.

2.Санах ойн ашиглалт:

- Divide and Conquer: Зөвхөн рекурсив дуудлагаар санах ой ашиглана.

- Динамик программчлал: Массив ашиглаж утгуудыг хадгална.

3.Цаг хугацааны төвөгтэй байдал:

- Divide and Conquer: O(2^n)
- Динамик программчлал: O(n)

4.Кодын боловсруулалт:

- Divide and Conquer: Рекурсив функц ашиглан бичихэд хялбар.
- Динамик программчлал: Давталт болон массив ашиглан бичихэд төвөгтэй байж болно.

3. Dynamic Programming vs Greedy

Жишээ: Хувийн үүргэвчний асуудал (Knapsack Problem)
Динамик Программчлалын Арга
Динамик программчлал нь том асуудлыг жижиг хэсгүүдэд хувааж, эдгээр хэсгүүдийг шийдэж, дахин нэгтгэн том асуудлыг шийддэг.
Код:

```
def knapsack_dynamic_programming(values, weights, capacity):
    n = len(values)
    dp = [[0 for _ in range(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]
# Жишээ өгөгдөл
values = [60, 100, 120]
weights = [10, 20, 30]
capacity = 50

# Динамик программчлал ашиглах
max_value_dp = knapsack_dynamic_programming(values, weights, capacity)
print(f"Динамик программчлалаар хамгийн их ашиг: {max_value_dp}")
```

Тайлбар:

1. **Инициализаци:** dp нь 2D жагсаалт бөгөөд dp[i][w] нь эхний i элементийн тусламжтайгаар жингийн хязгаар w хүрэхэд олж болох хамгийн их ашгийг илэрхийлнэ.
2. **Таблийг бөглөх:** Бүх элемент болон жингийн хязгаарыг давталт хийх:
    - Элементийн жин одоогийн жингийн хязгаарт хүрэх бол элементийг оруулах эсэхийг шийднэ.
    - Хэрэв биш бол элементийг алгасна**.**
3. **Үр дүн:** Хамгийн их ашиг dp[n][capacity]-д байрлана.

Greedy Algorithm

Greedy алгоритм нь хамгийн их ашиг олох алхмыг сонгож, дараагийн алхмыг дахин сонгодог.

Код:

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
    def knapsack_greedy(items, capacity):
        items.sort(key=lambda item: item.value/item.weight, reverse=True)
        total_value = 0

        for item in items:
            if capacity >= item.weight:
                capacity -= item.weight
                total_value += item.value
            else:
                total_value += item.value * (capacity / item.weight)
                break
        return total_value

        # Жишээ өгөгдөл
         items = [Item(60, 10), Item(100, 20), Item(120, 30)]
        capacity = 50

        # Greedy арга ашиглах
        max_value_greedy = knapsack_greedy(items, capacity)
        print(f"Греди аргаар хамгийн их ашиг: {max_value_greedy}")
```

Тайлбар:

1. **Инициализаци:** Эд зүйлсийн жагсаалтыг үүсгэж, тэдгээрийг үнэлгээ-ба-жин харьцаагаар эрэмбэлнэ.
2. **Сонголт:** Эрэмбэлэгдсэн эд зүйлсээс аль болох ихийг үүргэвчиндээ оруулна.
3. **Үр дүн:** Сонгосон эд зүйлсийн нийт үнэ цэнэ.

Ялгаа болон Шинжилгээ

1. **Хандлага:**

- Динамик Программчлал: Бүх боломжит шийдлүүдийг шалгаж, хамгийн их ашгийг олох.
- Греди Алгоритм: Одоогийн хамгийн сайн шийдлийг сонгож, эцсийн шийдлийг олох.

2. **Цаг хугацааны төвөгтэй байдал:**

- Динамик Программчлал: $O(n \times capacity)$
- Греди Алгоритм: $O(n \log n)$ (эрэмбэлэхээс болоод)

3. **Санах ойн төвөгтэй байдал:**

- Динамик Программчлал: $O(n \times capacity)$

- Греди Алгоритм: O (1) (эсвэл O (n) эд зүйлсийн жагсаалтыг хадгалахад)

4. **Оптимал байдал:**
- Динамик Программчлал: Хамгийн их ашиг олох шийдлийг баталгаажуулдаг.
- Греди Алгоритм: Хамгийн их ашиг олох шийдлийг баталгаажуулахгүй, гэхдээ илүү хурдан, хэрэгжүүлэхэд хялбар.

5. **Тохиромж:**
- Динамик Программчлал: Бүх боломжит шийдлүүдийг шалгах шаардлагатай асуудлуудад тохиромжтой.
- Греди Алгоритм: Local хамгийн сайн сонголтууд глобал хамгийн сайн шийдэлд хүргэдэг асуудлуудад тохиромжтой**.**

Ашигласан эх сурвалжууд :

- **Divide and Conquer Algorithm - GeeksforGeeks**
- **Greedy Algorithms - GeeksforGeeks**
- **Dynamic Programming or DP - GeeksforGeeks**
- **Recursion in Python - GeeksforGeeks**