ШИНЖЛЭХ УХААН ТЕХНОЛОГИЙН ИХ СУРГУУЛЬ

Мэдээлэл холбооны технологийн сургууль

БИЕ ДААЛТЫН ТАЙЛАН

Алгоритм шинжилгээ ба зохиомж (F.CSM301) 2024-2025 оны хичээлийн жилийн намар

Хичээл заасан багш: Д. Батмөнх

Бие даалт гүйцэтгэсэн: Э. Мичидмаа B221960015

Улаанбаатар, 2024

1. Divide-and-Conquer

Divide-and-Conquer бол асуудлыг жижиг дэд асуудлуудад хувааж, тус бүрийг бие даан шийдвэрлэсний дараа эдгээр шийдлийг нэгтгэн үндсэн асуудлын хариуг гаргах алгоритмын арга юм.

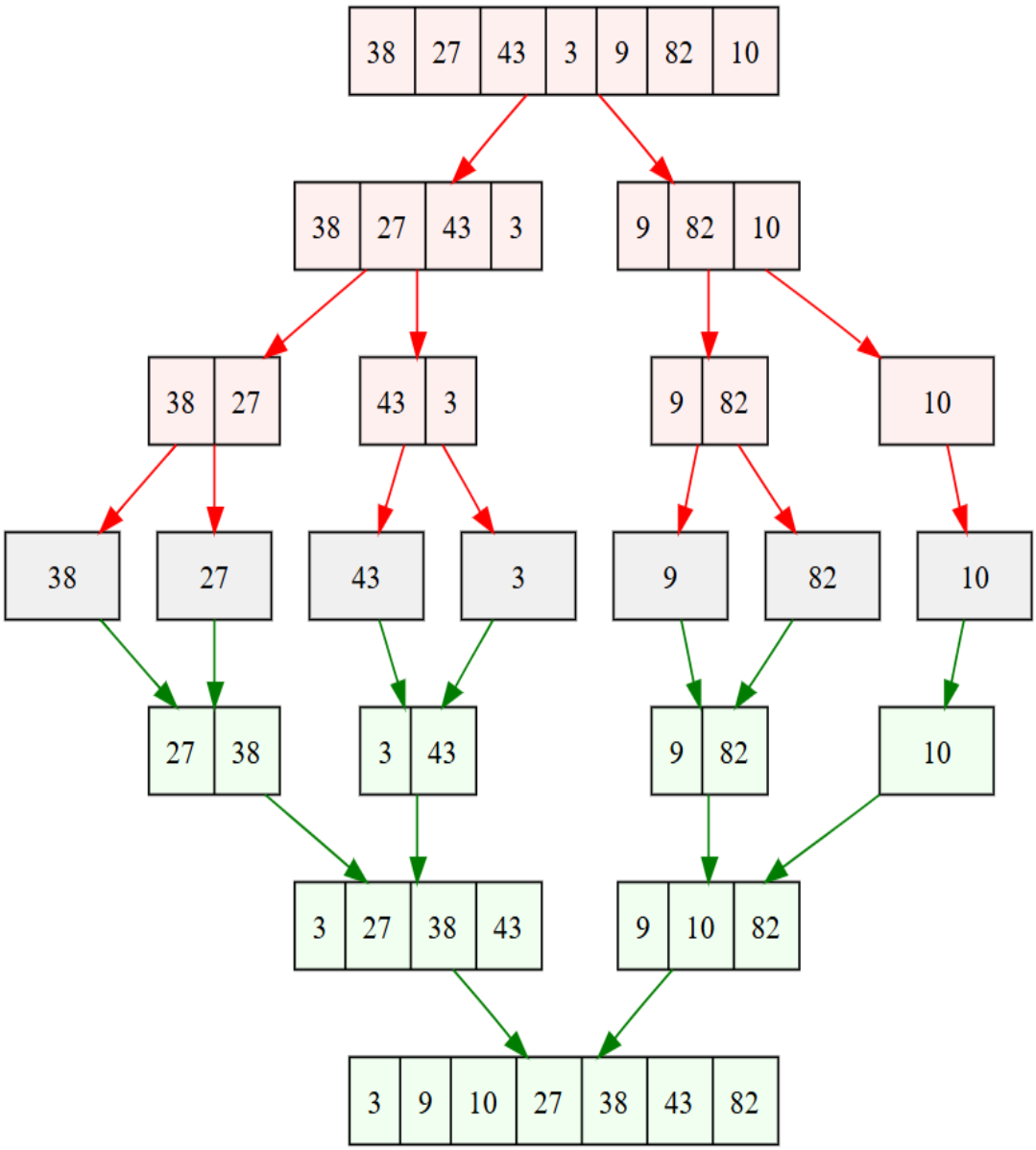
Энэ нь ихэвчлэн 3 үе шаттай:

1. **Хуваах (divide):** Том асуудлыг жижиг, хоорондоо хамааралгүй, илүү энгийн дэд асуудлуудад хуваана. Энэ шатанд асуудлыг хуваах дүрмийг тодорхойлох нь чухал.
2. **Шийдэх (conquer):** Хуваасан жижиг дэд асуудлуудыг тус бүрд нь шийднэ. Хэрэв дэд асуудал хангалттай жижиг болсон бол шууд шийдвэрлэнэ, эсвэл дахин хувааж шийддэг.
3. **Нэгтгэх (combine):** Дэд асуудлуудын шийдлийг нэгтгэж анхны үндсэн асуудлын хариуг гаргана. Энэ шатанд нэгтгэх аргачлал оновчтой байх шаардлагатай.

Жишээ:

- **Merge Sort**-д массивыг хоёр хувааж (divide), тус бүрийг эрэмбэлж (conquer), дараа нь эрэмбэлсэн массивуудыг нэгтгэдэг (combine).

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
                        if L[i] < R[j]:
                                arr[k] = L[i]
                                i += 1
                        else:
                                arr[k] = R[j]
                                j += 1
                        k += 1

        while i < len(L):
                        arr[k] = L[i]
                        i += 1
                        k += 1

        while j < len(R):
                        arr[k] = R[j]
                        j += 1
                        k += 1
```


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

2. Dynamic programming

Dynamic programming бол томоохон асуудлыг жижиг дэд асуудлуудад хувааж, тэдгээрийн давхардсан тооцоог хадгалж, дахин ашиглах замаар илүү оновчтой шийдвэр гаргах аргыг хэлнэ. Энэ нь ихэвчлэн **санах ой (memoization)** эсвэл **давталт (tabulation)** ашигладаг.

1. Санах ой (Memoization)

Тодорхойлолт: Санах ой гэдэг нь Dynamic Programming-ийн топ-даун (top-down) аргачлал бөгөөд асуудлыг дахин давтагдах дэд асуудлуудад хувааж, тэдгээрийн шийдлийг санах ойд хадгалж, ирээдүйд дахин ашигладаг арга юм.

Ажиллах зарчим:

1. **Рекурсив (давтагддаг) шийдэл: Асуудлыг дахин давтагддаг дэд асуудлуудад хуваана.**
2. **Хадгалах:** Шийдсэн дэд асуудлын хариуг санах ойд (жишээ нь, массив эсвэл хүснэгт) хадгална.
3. **Дахин ашиглах:** Хэрэв тухайн дэд асуудал дахин шаардлагатай бол дахин тооцоолохгүй, хадгалагдсан хариуг шууд ашиглана.

Жишээ: Fibonacci тооны дараалал тооцоолохдоо:

```
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```

Энд memo гэдэг нь тооцсон үр дүнг хадгалдаг санах ой юм. Энэ нь дахин давтагддаг тооцооллыг алгасаж, алгоритмын гүйцэтгэлийг сайжруулдаг

2. Давталт (Tabulation)

Тодорхойлолт: Давталт нь Dynamic Programming-ийн боттом-ап (bottom-up) аргачлал бөгөөд дэд асуудлуудыг аль хэдийн шийдсэнээс эхлэн илүү том дэд асуудлуудыг дараалан шийдэж, шийдлүүдийг хүснэгт буюу массивд хадгалдаг арга юм.

Ажиллах зарчим:

- **Дараалан шийдэх:** Асуудлыг хамгийн бага хэмжээндээс эхлэн шийдэж, илүү том дэд асуудлуудыг дараалан шийднэ.
- **Хадгалах:** Шийдсэн дэд асуудлын хариуг хүснэгт буюу массивд хадгална.
- **Хэрэглэх:** Хадгалагдсан хариуг ашиглан илүү том асуудлын шийдлийг гаргана.

Жишээ: Fibonacci тооны дараалал тооцоолохдоо:

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

Энд fib массив нь Fibonacci тооны хариуг хадгалдаг бөгөөд хамгийн бага dэд асуудлаас эхлэн илүү том dэд асуудлуудыг дараалан шийддэг.

3. Greedy algorithms

Greedy алгоритм бол тухайн мөчид хамгийн оновчтой гэж харагдаж буй сонголтыг хийж, асуудлыг алхам алхмаар шийдэх арга юм. Энэ нь үргэлж глобал оновчтой шийдэлд хүрдэггүй ч, тодорхой нөхцөлд хамгийн оновчтой шийдлийг гаргаж чаддаг. Greedy алгоритмуудын жишээ:

1. **Хамгийн ойрын хөрш алгоритм (Nearest Neighbor Algorithm):**

Аяллын маршрут төлөвлөлтөд ашиглагддаг. Ойрын хөршийг сонгож явсаар бүх цэгийг нэг удаа дамжина.

```
def nearest_neighbor(graph, start):
    visited = set([start])
    path = [start]
    current = start
    total_cost = 0

    while len(visited) < len(graph):
        next_node = None
        min_cost = float('inf')
        for neighbor, cost in graph[current]:
```

```
            if neighbor not in visited and cost < min_cost:
                min_cost = cost
                next_node = neighbor

        visited.add(next_node)
        path.append(next_node)
        total_cost += min_cost
        current = next_node

    return path, total_cost
# Жишээ граф
graph = {
    0: [(1, 10), (2, 15), (3, 20)],
    1: [(0, 10), (2, 35), (3, 25)],
    2: [(0, 15), (1, 35), (3, 30)],
    3: [(0, 20), (1, 25), (2, 30)],
}
path, cost = nearest_neighbor(graph, 0)
print("Маршрут:", path)
print("Нийт зардал:", cost)
```

Үр дүн:

Маршрут: `[0, 1, 3, 2]`

Нийт зардал: `65`

2. **Зоосны асуудал (Coin Change Algorithm):**

Өгөгдсөн зооснуудыг ашиглан тодорхой дүнг бүрдүүлэх хамгийн бага зоосны тоог олох.

```
def coin_change(coins, amount):
    coins.sort(reverse=True)  # Том зооснооc эхлэх
    result = []
    for coin in coins:
        while amount >= coin:
            amount -= coin
            result.append(coin)

    if amount == 0:
        return result
    else:
        return "Шийдэл байхгүй"

# Жишээ зоос
coins = [10, 5, 1]
amount = 18
print("Шийдэл:", coin_change(coins, amount))
```

Үр дүн:

Шийдэл: `[10, 5, 1, 1, 1]`

4. Recursion vs Divide-and-Conquer

Шинж чанар	Recursion	Divide-and-Conquer
Тодорхойлолт	Өөрийгөө дахин дуудаж асуудлыг шийдвэрлэх арга.	Асуудлыг жижиг хэсгүүдэд хувааж, тус тусад нь шийдэж, хариуг нэгтгэдэг стратеги.
Гол зарчим	Нэг том асуудлыг жижиг хэсгүүдэд хуваан, бүрэн шийдвэрлэх.	Асуудлыг хуваах (Divide), тус тусад нь шийдвэрлэх (Conquer), нэгтгэх (Combine).
Хэрэглээ	Давтагдах асуудлыг шийдэхэд тохиромжтой.	Асуудлыг хувааж, хурдан шийдлийг олох шаардлагатай үед.
Жишээ	Фибоначчийн тоо олох.	Merge Sort, Quick Sort, Binary Search зэрэг алгоритмууд.
Давуу тал	Алгоритмын бичихэд хялбар, ойлгоход амар.	Хуваах замаар том асуудлыг илүү хурдтай шийддэг.
Сул тал	Илүү их санах ой зарцуулдаг, давтагдах үйлдэлтэй үед үр ашиг багатай.	Хуваалт, нэгтгэл дээр төвөгтэй ажиллагаа шаардагдаж болно.

1. Recursion: Фибоначчийн тоо олох

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(5))  # Үр дүн: 5
```

Фибоначчийн тоог олохдоо өөрийгөө дахин дуудаж, n-1 ба n-2 дуудлагаар шийдвэрлэнэ. Жижиг утгуудыг давтан тооцоолно.

Сул тал:

Давхардсан тооцоолол ихтэй учир удаан ажиллаж, санах ой их зарцуулна.

2. Divide-and-Conquer: Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
```

```
        merge_sort(L)  # Зүүн хэсгийг хуваана
        merge_sort(R)  # Баруун хэсгийг хуваана

        i = j = k = 0

        # Хоёр массивыг нэгтгэн эрэмбэлэх
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

arr = [6, 3, 8, 5, 2]
merge_sort(arr)
print(arr)  # Үр дүн: [2, 3, 5, 6, 8]
```

Divide-and-Conquer нь том асуудлыг жижиг хэсгүүдэд хувааж, хурдан шийдвэрлэхэд ашиглагддаг. Илүү үр ашигтай бөгөөд том өгөгдлийг хурдан эрэмбэлнэ.

5. Divide-and-Conquer vs Dynamic Programming

Divide-and-Conquer:

- Асуудлыг жижиг хэсгүүдэд хувааж, тус бүрийг шийдээд, дараа нь эдгээр хэсгүүдийн шийдлийг нэгтгэдэг.
- Хэрэв асуудал нь дахин давтагдахгүй, заавал дахин тооцоолох шаардлагагүй бол энэ аргыг ашигладаг.
- Жишээ нь: Merge Sort, Quick Sort.

Dynamic Programming:

- Энэ арга нь асуудлыг жижиг хэсгүүдэд хувааж, тухайн хэсгүүдийн шийдлийг хадгалаад, дараа нь эдгээр шийдлүүдийг ашиглан илүү том асуудлыг шийддэг.
- Давтагдах асуудлуудыг дахин тооцоолохоос зайлсхийхээр өмнө нь шийдсэн хэсгүүдийг хадгалдаг.
- Жишээ нь: Fibonacci тоо, Longest Common Subsequence.

Харьцуулбал:

> **Divide-and-Conquer** нь асуудлыг бие даасан хэсгүүдэд хувааж, эдгээрийг тусад нь шийддэг, харин **Dynamic Programming** нь дахин тооцоолохоос зайлсхийхийн тулд шийдлийг хадгалан, дахин ашигладаг.

6. Dynamic Programming vs Greedy Algorithms

Dynamic Programming (DP) ба **Greedy Algorithms** хоорондын харьцуулалтаар, эдгээр хоёр арга нь шийдлийг олж авахад ашиглагддаг боловч тэдгээрийн үндсэн ялгаа нь хандлага, стратеги болон нөхцөл байдлын хамаарлаас үүдэлтэй.

Харьцуулалт:

1. **Шийдвэр гаргалт**:
    - **DP** нь бүх боломжит шийдлүүдийг харж, хамгийн сайн шийдлийг сонгохыг зорьдог. Энэ нь том асуудлыг жижиг дэд асуудлуудад хувааж, эдгээр дэд асуудлуудын шийдлийг хадгалж, хамгийн сайн үр дүнд хүрэхийг зорьдог.
    - **Greedy** нь зөвхөн одоогийн хамгийн сайн шийдлийг сонгодог. Энэ нь тухайн мөчид хамгийн их ашиг олохыг зорьж, зөвхөн тухайн алхам дээрх хамгийн сайн шийдлийг хайдаг.
2. **Хэрэглээ**:
    - **DP** нь давтагдах дэд асуудлуудтай, нөхцөл байдал нь илүү төвөгтэй болон олон урт шийдлүүдтэй байдаг асуудлуудад тохиромжтой. Мөн хамгийн сайн шийдлийг олж авахад чухал байдаг.
    - **Greedy** нь асуудал нь илүү энгийн, цаг хугацааны хязгаарлалтай, зөвхөн одоогийн хамгийн сайн шийдлийг шаарддаг тохиолдолд хамгийн үр ашигтай байдаг.
3. **Нийлмэл байдал**:
    - **DP** нь ихэнх тохиолдолд өндөр цагийн болон санах ойн зардалтай байдаг. Учир нь бүх дэд асуудлуудыг хадгалж, дахин ашиглах хэрэгтэй.
    - **Greedy** нь илүү хурдан, бага зардалтай бөгөөд тодорхой шийдлийг шууд олж чаддаг. Гэхдээ энэ нь бүх асуудалд үр дүнтэй биш.
4. **Шийдлийн чанар**:
    - **DP** нь бүх боломжит шийдлүүдийг зөв гаргаж, хамгийн сайн үр дүнд хүрэх боломжтой. Тиймээс энэ нь бүрэн шийдэл өгдөг.
    - **Greedy** нь зарим үед зөвхөн оройн хамгийн сайн шийдлийг олдог ч бүхэлд нь хамгийн сайн шийдэл биш байж болно.

Дүгнэлт:

- **Dynamic Programming** нь илүү нарийн төвөгтэй, олон давтагдах дэд асуудлуудтай асуудлуудын хувьд илүү үр дүнтэй, бүрэн шийдлийг олж чаддаг. Гэхдээ энэ нь илүү олон тооцоолол, өндөр цагийн зардал шаарддаг.
- **Greedy Algorithms** нь хурдан, бага зардалтай, тодорхой нөхцөлд ашиглахад тохиромжтой. Гэхдээ бүх асуудлуудыг шийдэхэд үр дүнтэй биш бөгөөд энэ нь зөвхөн хамгийн сайн шийдлийг олдоггүй.

Иймээс, **Dynamic Programming** нь илүү өргөн хүрээтэй, нарийн асуудлуудад тохиромжтой бол **Greedy Algorithms** нь хурдтай, энгийн шийдлүүдтэй асуудлуудын хувьд илүү тохиромжтой арга юм.

Ашигласан эх сурвалж:

1. https://en.wikipedia.org/wiki/Greedy_algorithm
2. https://en.wikipedia.org/wiki/Dynamic_programming
3. https://www.programiz.com/dsa/divide-and-conquer
4. https://www.geeksforgeeks.org/python-program-for-coin-change/
5. https://www.geeksforgeeks.org/divide-and-conquer/
6. https://onlinenotepad.org/notepad