Divide-and-Conquer

Энэхүү алгоритм нь төвөгтэй асуудлыг жижиг, илүү удирдаж болох хэсгүүдэд хуваааж, тус бүрийг бие даан шийдэж, дараа нь шийдлүүдийг нэгтгэн анхны асуудлыг шийдэх арга зүй юм. Энэ нь компьютерын шинжлэх ухаан, математикийн салбарт өргөнөөр ашиглагддаг алгоритмын аргачлал юм. Divide and conquer алгоритмыг **Divide, Conquer, Merge** гэсэн 3 хэсэгт хуваана.

Divide:

- Анхны асуудлыг жижиг дэд асуудлуудад хуваана.
- Бүх дэд асуудал нь нийт асуудлын нэг хэсгийг илэрхийлэх ёстой.
- Асуудлыг цаашид хуваах боломжгүй хүртэл хуваах нь гол зорилго юм.

Conquer:

- Жижиг дэд асуудал бүрийг бие даан шийднэ.
- Хэрэв дэд асуудал хангалттай жижиг бол (ихэвчлэн "суурь тохиолдол" гэж нэрлэдэг), цаашид дахин хуваахгүйгээр шууд шийднэ.
- Эдгээр дэд асуудлын шийдлийг бие даан олох нь зорилго юм.

Merge:

- Бүхэл асуудлын эцсийн шийдлийг олохын тулд дэд асуудлуудыг нэгтгэнэ.
- Жижиг дэд асуудлуудыг шийдсэний дараа тэдгээрийн шийдлүүдийг дахин хослуулан илүү том асуудлын шийдлийг олно.
- Дэд асуудлуудын шийдлээс анхны асуудлын шийдлийг гаргаж авах нь гол зорилго юм.

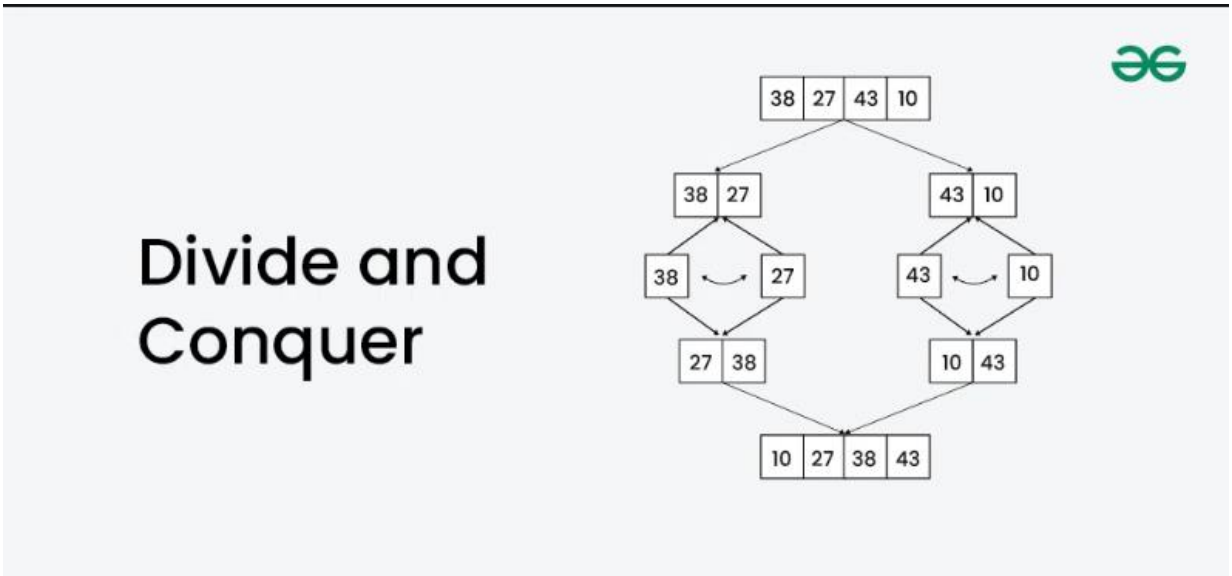

Figure 1. Divide and conquer(Merge sort)

Жишээ: Merge Sort

Merge sort нь массивыг хоёр хэсэгт хувааж, тус бүрийг нь эрэмбэлж, дараа нь хоёр хэсгийг нэгтгэж эцсийн эрэмбэлэгдсэн массивыг гаргана. Энэ нь асуудлыг хувааж шийдээд, дараа нь нэгтгэх классик жишээ юм.

```
def merge_sort(arr):
    if len(arr) > 1:
        # 1. Divide
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort both halves
        merge_sort(left_half)
        merge_sort(right_half)

        # 2. Merge
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
        # Check for any remaining elements
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

B221910011 Ц.Тэмүүжин

Dynamic Programming (DP)

Динамик программчлал гэдэг нь төвөгтэй асуудлуудыг жижиг дэд асуудлуудад хувааж шийдэх арга бөгөөд математик болон компьютерын шинжлэх ухаанд ашиглагддаг. Дэд асуудал бүрийг зөвхөн нэг удаа шийдэж, үр дүнг нь хадгалах замаар дахин тооцооллоос зайлсхийж, олон төрлийн асуудлыг үр ашигтайгаар шийдэж чадна. Dynamic programming нь математик, компьютерын шинжлэх ухаанд нийлмэл бодлогуудыг энгийн дэд бодлого болгон задлахад ашигладаг арга юм. Дэд асуудал бүрийг зөвхөн нэг удаа шийдэж, үр дүнгээ хадгалснаар илүү их тооцоолол хийхээс зайлсхийж, өргөн хүрээний асуудлуудыг илүү үр дүнтэй шийдвэрлэх боломжтой болгодог.

Динамик программчлал (DP) хэрхэн ажилладаг вэ?

- Дэд асуудлуудыг тодорхойлох: Гол асуудлыг жижиг, бие даасан дэд асуудлуудад хуваана.
- Шийдлүүдийг хадгалах: Дэд асуудал бүрийг шийдээд, шийдлийг хүснэгт эсвэл массивт хадгална.
- Шийдлүүдийг нэгтгэх: Хадгалсан шийдлүүдийг ашиглан гол асуудлын шийдлийг гаргаж авна.
- Давхардлаас зайлсхийх: Шийдлүүдийг хадгалах замаар динамик программчлал нь дэд асуудал бүрийг зөвхөн нэг удаа шийдэх боломжтой болгож, тооцооллын цагийг багасгадаг.

Динамик программчлалыг дараах хоёр аргаар хэрэгжүүлж болно:

Top-Down арга (Memoization)

- Top-Down аргыг мемоизаци гэж нэрлэх бөгөөд эцсийн шийдлээс эхлэн дэд асуудлууд руу шат дараалалтайгаар хуваадаг. Давхардсан тооцооллоос зайлсхийхийн тулд шийдэгдсэн дэд асуудлуудын үр дүнг мемоизацийн хүснэгтэд хадгалдаг.

Bottom-Up арга (Tabulation)

- Bottom up аргыг табуляц гэж нэрлэх бөгөөд хамгийн жижиг дэд асуудлуудаас эхэлж, аажмаар эцсийн шийдэл руу тооцоолдог. Давхардсан тооцооллоос зайлсхийхийн тулд шийдэгдсэн дэд асуудлуудын үр дүнг хүснэгтэд хадгална.

Жишээ: Фибоначчийн дараалал

Фибоначчийн тооны дараллыг тооцоолоход DP ашиглан өмнөх үр дүнг санах ойд хадгалж, дахин тооцоололгүйгээр шууд хариуг гаргаж болно. Рекурсын аргаар хийх үед нэг тоог олон удаа дахин тооцоолох шаардлага гардаг ч DP ашиглах үед үүнийг зайлсхийж чадна.

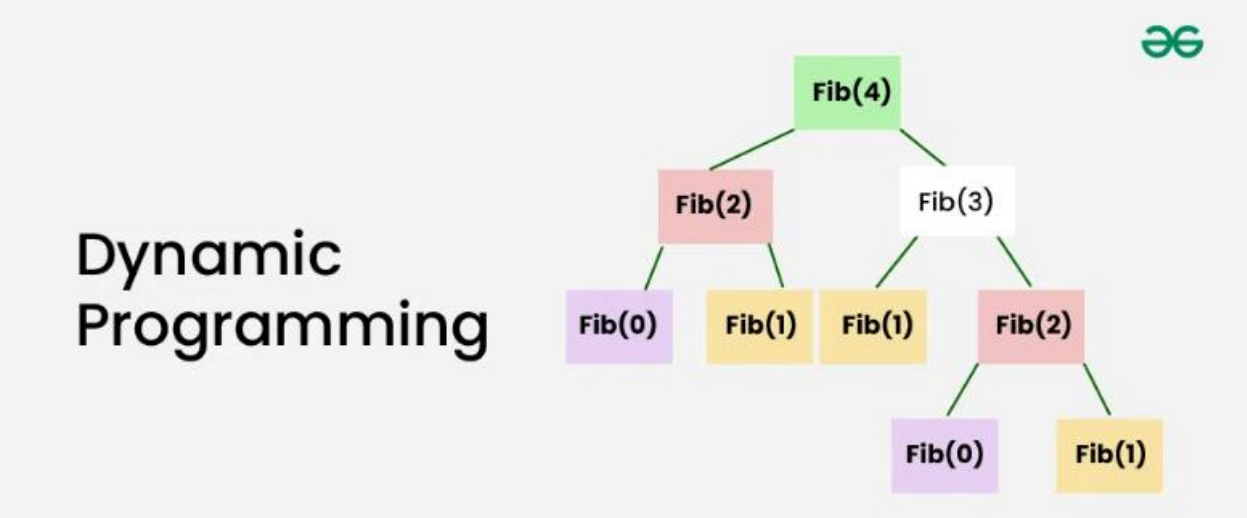

Figure 2. dynamic programming(fibonacci)

```
def fibonacci_dp(n):
    fib = [0] * (n + 1)
    fib[0], fib[1] = 0, 1

    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]

    return fib[n]
```

Greedy Algorithms

Greedy алгоритм гэдэг нь нийт шийдэлд хүрэх найдвартайгаар алхам тутамд local орчинд хамгийн сайн шийдлийг сонгодог алгоритмын ангилал юм. Эдгээр алгоритмуудад шийдвэрийг тухайн үеийн мэдээлэл дээр үндэслэн гаргадаг бөгөөд цаашдын үр дагаврыг харгалзан үздэггүй. Гол санаа нь алхам тутамд хамгийн боломжит сайн сонголтыг сонгож, заримдаа хамгийн оновчтой шийдэл биш боловч олон асуудалд хангалттай сайн шийдлийг олох явдал юм. Алдартай шунахай алгоритмуудын жишээнд Хувьсах үүргэвчийн асуудал (Fractional Knapsack), Дейкстрагийн алгоритм (Dijkstra's algorithm), Крускалын алгоритм (Kruskal's algorithm), Хаффманы кодчлол (Huffman coding), Примийн алгоритм (Prim's Algorithm) зэрэг орно.

```
# Greedy approach for Fractional Knapsack Problem
class Item:
    def __init__(self, value, weight):
        self.value = value
```

```
        self.weight = weight
        self.ratio = value / weight
def fractional_knapsack(items, capacity):
    # Sort items by value-to-weight ratio
    items.sort(key=lambda x: x.ratio, reverse=True)


    total_value = 0
    for item in items:
        if capacity > 0:
            if item.weight <= capacity:
                # Take the full item
                total_value += item.value
                capacity -= item.weight
            else:
                # Take the fraction of the item
                fraction = capacity / item.weight
                total_value += item.value * fraction
                capacity = 0


    return total_value
```

Recursion vs Divide-and-Conquer

Recursion нь функцийн өөрийгөө дахин дуудаж, асуудлыг жижгэрүүлж шийдэх арга юм. Энэ аргын хүрээнд нэгтгэх алхам заавал байх шаардлагагүй. Харин divide and conquer нь асуудлыг бие даасан дэд хэсгүүдэд хувааж, тус бүрийг шийдэх замаар эцсийн хариуг гаргадаг. Энэ аргын хувьд хуваасан дэд асуудлууд нь хоорондоо бие даасан байх ёстой, ингэснээр тэдгээрийг тус бүр шийдээд нэгтгэж болно. Merge sort зэрэг алгоритм нь divide and conquer аргачлалыг ашиглаж, дэд жагсаалтуудыг нэгтгэж эцсийн жагсаалтыг гаргадаг. Recursion нь хялбар, хурдан хөгжүүлж болох боловч их хэмжээний давталт үүсэх үед гажуудал гарч болзошгүй. Recursion нь асуудлыг давтсан байдлаар шийддэг бол divide and conquer нь хувааж, шийдэж, нэгтгэх процессоор онцлогтой.

Divide-and-Conquer vs Dynamic Programming

Divide-and-Conquer нь асуудлыг жижиг, бие даасан дэд асуудлуудад хувааж, тус бүрийг шийдэж, дараа нь шийдлүүдийг нэгтгэх замаар ажилладаг. Энэ нь ихэвчлэн дэд асуудлууд нь хоорондоо бие даан, хамааралгүй байдаг бөгөөд дэд асуудлуудыг шийдэхийн тулд ихэвчлэн дахин дуудаж, тооцоолох шаардлага гардаг.

Dynamic Programming нь дэд асуудлуудын шийдлүүдийг дахин ашиглах, хадгалах механизмыг ашиглан асуудлыг шийдэх замаар ажилладаг. Энэ нь дэд асуудлууд хоорондоо хамааралтай байх тохиолдолд хамгийн үр дүнтэй бөгөөд алгоритм нь дэд асуудлуудын шийдлүүдийг хадгалж, дахин тооцоолохоос зайлсхийдэг.

Dynamic programing	Divide and conquer
Динамик программчлалд шийдвэр гаргах олон дарааллыг бий болгож, бүх давхардсан дэд тохиолдлуудыг авч үздэг.	Энэ техникт асуудлыг жижиг дэд асуудалд хуваадаг. Эдгээр дэд асуудлуудыг бие даан шийддэг. Эцэст нь өгөгдсөн асуудлын эцсийн шийдлийг гаргахын тулд дэд асуудлын бүх шийдлүүдийг нэгтгэдэг.
Динамик программчлалын хувьд шийдлүүдийн давхардлаас бүрэн зайлсхийдэг.	Энэ аргын хувьд дэд шийдлүүдийн давхардлыг үл тоомсорлодог, өөрөөр хэлбэл давхар шийдлийг олж авах боломжтой.
Динамик программчлал нь "Divide and conquer" техникээс илүү үр дүнтэй байдаг.	Шийдлүүдийг дахин боловсруулах шаардлагатай тул "Divide and conquer" арга нь динамик программчлалаас бага үр дүнтэй байдаг.
Энэ нь рекурсив бус арга юм.	Энэ нь рекурсив арга юм.
Асуудлыг шийдвэрлэхдээ bottom up аргыг ашигладаг.	Асуудлыг шийдвэрлэхдээ top down аргыг ашигладаг.

Dynamic Programming vs Greedy

Dynamic Programming нь асуудлыг дэд асуудлуудад хувааж, дэд асуудлуудын шийдлийг хадгалах замаар нийт шийдлийг олохыг зорьдог. Энэ нь давтагддаг дэд асуудлуудтай (overlapping subproblems) болон хамааралтай шийдлүүдтэй (optimal substructure) асуудлуудад хамгийн үр дүнтэй байдаг. Харин Greedy алгоритм нь одоогийн нөхцөл байдлыг харгалзан, хамгийн сайн харагдаж буй шийдлийг шууд сонгох замаар ажилладаг. Энэ нь хурдан бөгөөд ихэнх тохиолдолд илүү бага санах ой шаарддаг боловч зөв шийдэлд хүрэхгүй байж болох юм. Dynamic Programming нь илүү нарийвчилсан, ойлгомжтой шийдэлд хүргэдэг бол Greedy алгоритм нь илүү хурдан, олон тооны асуудлыг шийдэхэд амар байдаг.

Онцлог	Greedy	Dynamic programming
Оновчтой байдал	Үргэлж оновчтой шийдлийг гаргаж чаддаггүй.	Хэрэв асуудал нь оновчтой байх зарчмыг харуулсан бол оновчтой шийдлийг баталгаажуулдаг.
Дэд асуудал дахин ашиглах	Дэд асуудлын шийдлийг дахин ашигладаггүй.	Давхардсан дэд асуудлын шийдлүүдийг дахин ашигладаг.
Нарийн төвөгтэй байдал	Ихэвчлэн илүү хялбар бөгөөд хэрэгжүүлэхэд илүү хурдан байдаг.	Хэрэгжүүлэхэд илүү төвөгтэй, удаан байж болно.
Application	Local оновчлол нь Global оновчлолд хүргэдэг асуудлуудад тохиромжтой.	Давхардсан дэд асуудал, оновчтой дэд бүтэцтэй асуудлуудад тохиромжтой.
Example	Хамгийн бага хүрээтэй мод, хамгийн богино зам алгоритмууд.	Фибоначчийн дараалал, хамгийн урт нийтлэг дэд дараалал.

Ашигласан материал:

- (https://www.geeksforgeeks.org/divide-and-conquer/, n.d.)
- (https://www.geeksforgeeks.org/dynamic-programming/, n.d.)
- (https://www.geeksforgeeks.org/greedy-algorithms/?ref=shm, n.d.)
- (https://www.javatpoint.com/dynamic-programming-vs-divide-and-conquer, n.d.)
- (https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/, n.d.)