

ШИНЖЛЭХ УХААН ТЕХНОЛОГИЙН ИХ СУРГУУЛЬ
Мэдээлэл, холбооны технологийн сургууль



Бие даалт

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

Шалгасан багш:
Гүйцэтгэсэн:

Д.Батмөнх
Ц.Болортуяа / В221910024 /

Улаанбаатар хот
2024 он

Агуулга

Удиртгал	3
Divide and Conquer	4
Ерөнхий ойлголт	4
Давуу тал.....	4
Сул тал	5
Жишээ бодлого.....	6
Dynamic Programming	7
Ерөнхий ойлголт	7
Dynamic Programming-ийн үндсэн хоёр арга.....	7
Memoization (Top-down approach)	7
Tabulation (Bottom-up approach).....	7
Давуу тал.....	7
Сул тал	8
Жишээ бодлого.....	8
Greedy Algorithms	9
Ерөнхий ойлголт	9
Давуу тал.....	10
Сул тал	10
Жишээ бодлого.....	10
Recursion vs Divide-and-Conquer	12
Divide-and-Conquer vs Dynamic Programming	12
Dynamic Programming vs Greedy	12

Удиртгал

Энэхүү бие даалтын хүрээнд Divide-and-Conquer, Dynamic Programming, Greedy Algorithm-ийн талаар судалж тэдгээрийг хэрхэн ажилладаг болон хоорондын ялгааг судалсан. Мөн эдгээрийг судлахдаа тус бүрийн ажиллагааг python code-оор туршсан.

Divide and Conquer

Ерөнхий ойлголт

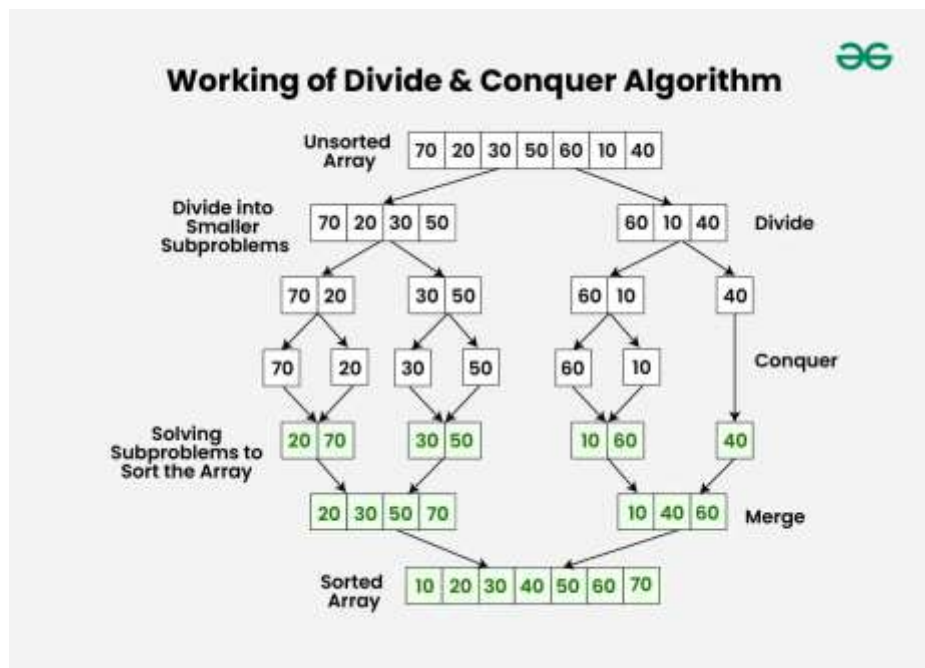
Divide and Conquer арга нь асуудлыг жижиг дэд хэсгүүдэд хуваан бие даан шийдвэрлэсний дараа түүнийгээ нэгтгэдэг. Гол санаа нь асуудлыг шууд шийдвэрлэхэд хялбар болох хүртэл жижиг дэд бодлогод хуваах явдал юм. Дэд хэсгүүдийн шийдлүүдийг олж авсны дараа тэдгээрийг нэгтгэж ерөнхий шийдлийг гаргана.

Divide and Conquer-ийг алгоритм нь хуваах (Divide), эзлэх (Conquer), нэгтгэх (Merge) гэсэн 3 хэсгээс бүрдэнэ.

Хуваах (Divide): Асуудлыг цаашид хуваагдах боломжгүй болтол нь дэд хэсгүүдэд хуваана. Дэд хэсгүүд бүр нь ерөнхий асуудлын нэг хэсгийг төлөөлдөг.

Эзлэх (Conquer): Хуваагдсан дэд хэсгүүд дэх асуудал бүрийг тус тусад нь шийднэ. Гол зорилго нь асуудал бүрийн шийдлийг бие даан олох юм.

Нэгтгэх (Merge): Дэд хэсгүүдийн үр дүнг нэгтгэж анхны асуудлын шийдлийг олно. Нэгтгэхдээ дэд хэсгүүдийн үр дүнг рекурсив байдлаар нэгтгэдэг.



Зураг 1: divide and conquer algorithm

Давуу тал:

Цогц асуудлыг шийдвэрлэх хялбар байдал:

- Том асуудлыг жижиг, амархан шийдэгдэх хэсгүүдэд хувааж, тус бүрийг шийдэх боломжтой болгодог.

Давтагдах чанар:

- Энэ арга нь рекурсив хэлбэрээр асуудлыг шийдэхэд тохиромжтой бөгөөд олон алгоритмд ашиглагддаг (жишээ нь: Merge Sort, Quick Sort, Binary Search).

Хурдасгах боломж:

- Том хэмжээний өгөгдөлд тохиромжтой. Параллель тооцоололд хуваагдсан хэсгүүдийг зэрэгцээ гүйцэтгэх боломжтой.

Уян хатан байдал:

- Янз бүрийн асуудлуудад ашиглах боломжтой бөгөөд өгөгдлийн бүтцээс хамаарал багатай.

Математикт суурилсан анализ:

- Divide and Conquer аргыг ашиглан алгоритмын цаг хугацааны нарийвчилсан дүн шинжилгээ хийхэд хялбар (жишээ нь: Master Theorem ашиглах).

Сул тал:

Санах ойны хэрэглээ өндөр:

- Рекурсив аргын үед санах ойд их хэмжээний стек ашигладаг. Энэ нь том хэмжээний өгөгдөлд асуудал үүсгэж болзошгүй.

Үр ашиггүй хуваалт:

- Зарим тохиолдолд хуваагдсан хэсгүүдийн дундах нэгдэл (merge) эсвэл харилцан үйлдэл хийх ажиллагаа хэт их цаг зарцуулж болзошгүй (жишээ нь: Merge Sort).

Тооцооллын давхардал:

- Зарим алгоритмуудын хувьд (жишээ нь: Fibonacci-рекурсив) хуваасан хэсгүүд давхардсан тооцоолол үүсгэж, цаг хугацааны үр ашиггүй байдал үүсгэж болно.

Хэрэгжилтийн нарийн төвөгтэй байдал:

- Зарим асуудлуудыг хувааж шийдэхэд тохиромжтой логик эсвэл загвар гаргахад төвөгтэй байж болно.

Жишээ бодлого:

Жишээ бодлогоор merge sort хийх аргыг python code дээр туршсан. Merge sort буюу нэгтгэн эрэмбэлэх арга нь divide and conquer аргын хамгийн энгийн алгоритм юм.

```
merge_sort.py X
D: > Bogi > Алгоритмын шинжилгээ > merge_sort.py > ...
1  def merge_sort(arr):
2      if len(arr) <= 1:
3          return arr
4
5      # Дундаж цэгийг олох
6      mid = len(arr) // 2
7
8      # Массивыг хоёр хэсэгт хуваах
9      left_half = merge_sort(arr[:mid])
10     right_half = merge_sort(arr[mid:])
11
12     # Хоёр хэсгийг нэгтгэх
13     return merge(left_half, right_half)
14
15 def merge(left, right):
16     sorted_array = []
17     i = j = 0
18
19     # Хоёр хэсгийг нэгтгэх (эрэмбэлсэн байдлаар)
20     while i < len(left) and j < len(right):
21         if left[i] < right[j]:
22             sorted_array.append(left[i])
23             i += 1
24         else:
25             sorted_array.append(right[j])
26             j += 1
27
28     # Үлдсэн элементүүдийг нэмэх
29     sorted_array.extend(left[i:])
30     sorted_array.extend(right[j:])
31
32     return sorted_array
33
34 arr = list(map(int, input("Массивын элементүүдийг оруулна уу: ").split()))
35 sorted_arr = merge_sort(arr)
36 print("Эрэмбэлэгдсэн массив:", sorted_arr)
37
```

Зураг 2: merge sort python code

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Bogi\Алгоритмын шинжилгээ> python3 .\merge_sort.py
Массивын элементүүдийг оруулна уу: 10 22 39 1 3 5 199
Эрэмбэлэгдсэн массив: [1, 3, 5, 10, 22, 39, 199]
PS D:\Bogi\Алгоритмын шинжилгээ> 
```

Зураг 3: merge sort python result

Dynamic Programming

Ерөнхий ойлголт

Динамик Программчлал нь массив эсвэл хүснэгт ашиглан асуудлыг хамгийн үр дүнтэй шийдэх аргын нэг юм. Энэ арга нь алдааг дахин давтах (overlapping subproblems) болон үр ашигтай хадгалалт (optimal substructure) гэсэн хоёр үндсэн зарчим дээр суурилдаг.

Давхардсан дэд асуудлууд (Overlapping Subproblems): Энэ нь алгоритм том асуудлын олон дэд асуудал буюу хэсгүүдийн шийдлийг хадгалж өөр адил асуудалтай хэсгүүдэд хадгалсан шийдлийг ашигладаг. Ингэснээр нэг л удаа тооцоолон, дахин дахин ашиглах боломжтой.

Оновчтой дэд бүтэц (Optimal Substructure): Энэ нь том асуудлын оновчтой үр дүнд хүрэхийн тулд дэд асуудлын оновчтой үр дүнг ашиглахыг хэлнэ.

Динамик програмчлал ажиллах зарчим нь divide and conquer аргатай адил томоохон асуудлыг жижиг дэд асуудлуудаар хуваадаг боловч, эдгээр дэд асуудлуудыг нэг удаа шийдэж, дараа нь эдгээр шийдлүүдийг хадгалж ашигладгаараа ялгаатай. Ингэснээр давтан тооцооллыг хийхгүй бөгөөд хурдан бөгөөд үр дүнтэй шийдлийг гаргаж чаддаг.

Dynamic Programming-ийн үндсэн хоёр арга:

❖ Memoization (Top-down approach):

- Рекурсив аргаар жижиг дэд асуудлуудыг шийдэж, эдгээр шийдлүүдийг хадгалж ашигладаг.
- Хэрэв дэд асуудлын шийдлийг дахин тооцоолох шаардлага гарвал, дахин тооцоолохгүйгээр хадгалсан утгыг шууд ашиглана.

❖ Tabulation (Bottom-up approach):

- Энэ арга нь хамгийн жижиг дэд асуудлаас эхлэн шийдэл гаргаж, бүх боломжит шийдлүүдийг эхнээс нь дэс дараалалтайгаар олох аргыг ашигладаг.
- Энд бид бүх шийдлийг массив эсвэл хүснэгтэд хадгалж, хамгийн том шийдлийг гаргадаг.

Давуу тал:

Цаг хугацааны хэмнэлт:

- Давтан тооцоолол хийхгүй учраас хурдан гүйцэтгэх боломжтой.

Санах ойн хадгалалт:

- Дэд асуудлуудын шийдлийг хадгалах, дахин ашиглах боломжтой болсноор санах ойг үр ашигтай ашиглана.

Сул тал:

Ой санамжийн хэрэглээ өндөр:

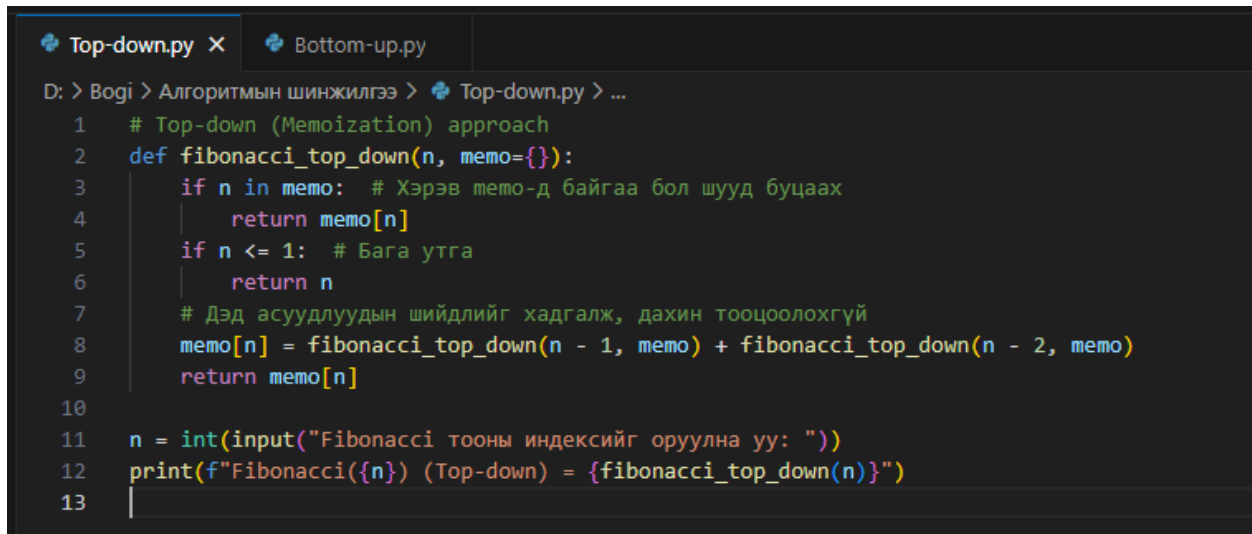
- Дэд асуудлуудыг хадгалахад их хэмжээний санах ой шаарддаг.

Гүйцэтгэх хурд удаашрах:

- Зарим тохиолдолд буферчлэлээс шалтгаалан хурд удааширч болно.

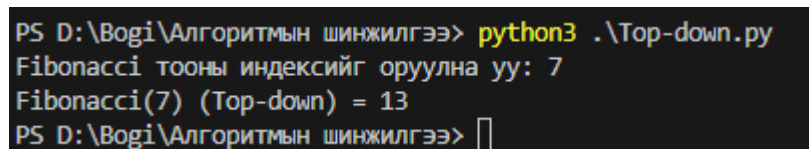
Жишээ бодлого:

Динамик програмчлалын хамгийн энгийн жишээ бол Fibonacci тоо юм. Энэ жишээг Memoization (Top-down) болон Tabulation (Bottom-up) аргууд дээр бодов.



```
Top-down.py X Bottom-up.py
D: > Bogi > Алгоритмын шинжилгээ > Top-down.py > ...
1  # Top-down (Memoization) approach
2  def fibonacci_top_down(n, memo={}):
3      if n in memo: # Хэрэв мемо-д байгаа бол шууд буцаах
4          return memo[n]
5      if n <= 1: # Бага утга
6          return n
7      # Дэд асуудлуудын шийдлийг хадгалж, дахин тооцоолохгүй
8      memo[n] = fibonacci_top_down(n - 1, memo) + fibonacci_top_down(n - 2, memo)
9      return memo[n]
10
11 n = int(input("Fibonacci тооны индексийг оруулна уу: "))
12 print(f"Fibonacci({n}) (Top-down) = {fibonacci_top_down(n)}")
13
```

Зураг 4: top-down аргын код



```
PS D:\Bogi\Алгоритмын шинжилгээ> python3 .\Top-down.py
Fibonacci тооны индексийг оруулна уу: 7
Fibonacci(7) (Top-down) = 13
PS D:\Bogi\Алгоритмын шинжилгээ>
```

Зураг 5: top-down аргын кодын үр дүн


```
Top-down.py Bottom-up.py X
D: > Bogi > Алгоритмын шинжилгээ > Bottom-up.py > ...
1  # Bottom-up (Tabulation) approach
2  def fibonacci_bottom_up(n):
3      if n <= 1: # Бага утга
4          return n
5      fib = [0] * (n + 1) # Fibonacci массив үүсгэж
6      fib[1] = 1
7      for i in range(2, n + 1): # Дараагийн тоог тооцоолох
8          fib[i] = fib[i - 1] + fib[i - 2]
9      return fib[n]
10
11 n = int(input("Fibonacci тооны индексийг оруулна уу: "))
12 print(f"Fibonacci({n}) (Bottom-up) = {fibonacci_bottom_up(n)}")
13
```

Зураг 6: bottom-up аргын код

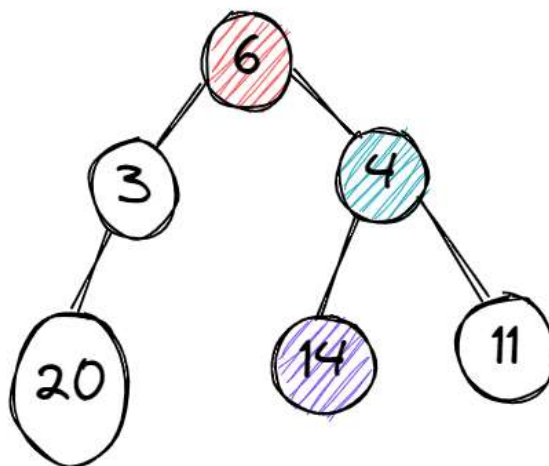
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\Bogi\Алгоритмын шинжилгээ> python3 .\Bottom-up.py
Fibonacci тооны индексийг оруулна уу: 10
Fibonacci(10) (Bottom-up) = 55
PS D:\Bogi\Алгоритмын шинжилгээ>
```

Зураг 7: bottom-up аргын кодын үр дүн

Greedy Algorithms

Ерөнхий ойлголт

Энэ алгоритм нь бий болох шийдлийн ерөнхий оновчлолыг харгалзахгүйгээр яг одоо оновчтой мэт санагдаж буй замыг сонгодог. Өөрөөр хэлбэл үр дүн ямар байхаас үл хамааран зөв гэж үзсэн замыг сонгоно.



Зураг 8: greedy algorithm example

Зураг 8-д үзүүлсэн схем нь хамгийн их утгатай тоог олох зорилготой. 6-г орой гэж үзвэл 3 болон 4 гэсэн утгаас аль нь их вэ гэдгийг шалгаж 4-г сонгоно. Түүний дараа 11 болон 14 утгуудаас 14-г сонгоно. Утгаа бол энэ схемийн хувьд 20 нь хамгийн их тоо боловч greedy algorithm-ийн зарчмаар 3 болон 4-н утгыг шалгах үед 4 нь их тоо гэж үзсэн учир үр дүн ямар байхаас үл хамааран тухайн үед зөв гэж үзсэн шийдвэрийг сонгож байна.

Давуу тал:

Энгийн бөгөөд хурдан:

- Ихэнх тохиолдолд эргэлзээтэй оролдлого хийхгүйгээр шууд шийдвэр гаргаж, хурдан шийдэл олох боломжтой.

Том өгөгдлийг хялбар шийдэх:

- Алгоритм нь олон жижиг асуудлыг шийдэхийн тулд түргэн шийдэлтэй байдаг.

Сул тал:

Алгоритмын чанар (Optimality):

- Бүх төрлийн асуудалд хамгийн сайн шийдлийг өгч чадахгүй.

Үр дүнг тооцдоггүй:

- Зөвхөн одоогийн нөхцөлд хамгийн сайн шийдлийг сонгоход анхаардаг тул үр дүнд нөлөөлөх хүчин зүйлсийг алгасдаг.

Жишээ бодлого:

Хамгийн бага зоосны тоог олох greedy algorithm-ийг python дээр бодов. Энэ нь 1, 5, 10, 25, 50, 100 дүнтэй зооснуудаар гараас өгөгдсөн дүнг хамгийн багадаа хэдэн зоосоор төлж болох тоог олох зорилготой.

```
min_num_coin.py X
D: > Bogi > Алгоритмын шинжилгээ > min_num_coin.py > ...
1  def greedy_coin_change(coins, total):
2      # Зоосны дүнг хамгийн ихээс хамгийн бага руу зөв дараалалд оруулна
3      coins.sort(reverse=True)
4
5      coin_count = 0 # Төлөх зоосны тоо
6      coin_usage = {}
7      for coin in coins:
8          if total == 0:
9              break
10         num_coins = total // coin # Тухайн зоосоор хэдэн удаа төлөхийг тооцоолно
11         if num_coins > 0:
12             coin_usage[coin] = num_coins # Зоосны тоо
13             coin_count += num_coins
14             total = total % coin
15
16     return coin_count, coin_usage
17
18 coins = [1, 5, 10, 25, 50, 100] # Зоосны дүн
19 total = int(input("Төлөх мөнгө оруулна уу: ")) # Төлөх мөнгө
20
21 result_count, result_usage = greedy_coin_change(coins, total)
22 print(f"Цөөн зоосоор төлөх тоо: {result_count}")
23 print("Зоосны хэрэглээ:")
24 for coin, usage in result_usage.items():
25     print(f"{coin} зоос: {usage} удаа")
26
```

Зураг 9: greedy algorithm code

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\Bogi\Алгоритмын шинжилгээ> python3 .\min_num_coin.py
Төлөх мөнгө оруулна уу: 403
Цөөн зоосоор төлөх тоо: 7
Зоосны хэрэглээ:
100 зоос: 4 удаа
1 зоос: 3 удаа
PS D:\Bogi\Алгоритмын шинжилгээ> █
```

Зураг 10: greedy algorithm code result

Recursion vs Divide-and-Conquer

Recursion: Асуудлыг өөрөө өөрийгөө дуудаж шийддэг (өөрийгөө дуудах функцийг ашиглана). Энэ нь жижгэрүүлж шийдэх арга бөгөөд олон шаттай нэгтгэл заавал шаардлагагүй. Жишээ: Factorial, Fibonacci тооцоолол.

Divide-and-Conquer: Асуудлыг дэд асуудалд заавал хувааж, тус тусад нь шийдээд эцэст нь үр дүнг нэгтгэдэг. Жишээ: Merge Sort, Binary Search.

Гол ялгаа нь Divide-and-Conquer нь "нэгтгэх" шаттай байдаг бол, Recursion нь энэ үе шатыг шаарддаггүй. Мөн recursion нь функц өөрийгөө дууддагт байгаа.

Divide-and-Conquer vs Dynamic Programming

Divide-and-Conquer: Асуудлыг дэд асуудлуудад хувааж, тэдгээрийг бие даан шийддэг. Дэд асуудлууд хоорондоо давхцахгүй. Нэг асуудал шийдсэн бол дахин ашиглах шаардлагагүй. Жишээ: Merge Sort, Binary Search.

Dynamic Programming: Асуудлыг дэд асуудалд хуваадаг боловч дэд асуудлууд давхцах боломжтой. Тиймээс нэг дэд асуудлын шийдлийг хадгалж, дахин ашигладаг (Memoization эсвэл Tabulation ашиглана). Жишээ: Fibonacci, Shortest Path (Dijkstra's, Floyd-Warshall).

Гол ялгаа: Dynamic Programming нь давхцсан дэд асуудлууд болон үр дүнг хадгалах шаардлагатай үед ашиглагддаг бол Divide-and-Conquer нь хамааралгүй дэд асуудлууд дээр тулгуурладаг. Мөн Dynamic Programming нь subproblems-ийг шийдсэн шийдлийг санах ой эсвэл хүснэгтэд хадгалдгаараа ялгаатай. Ингэж хадгалж өгснөөрөө Divide-and-Conquer-ээс арай бага resource шаардахаас гадна илүү хурдан ажиллах боломжтой.

Dynamic Programming vs Greedy

Greedy Algorithm нь тухайн мөчид хамгийн ашигтай шийдвэрийг сонгож, дараагийн алхмуудыг үргэлжлүүлэн шийддэг. Ихэвчлэн Dynamic Programming-аас хурдан ажилладаг, учир нь бүх хослолыг туршдаггүй. Жишээ: Fractional Knapsack, Minimum Spanning Tree (Prim's, Kruskal's), Dijkstra's Algorithm, Activity Selection Problem

Гол ялгаа: Dynamic Programming нь бүх боломжит сонголтыг туршиж, хамгийн тохиромжтой шийдлийг олдог, харин Greedy нь тухайн мөчид хамгийн ашигтай шийдлийг сонгодог. Dynamic Programming нь ихэвчлэн илүү нарийн шийдлийг өгдөг боловч удаан байж болно, харин Greedy нь хурдан боловч бүх тохиолдолд зөв шийдэл өгдөггүй. Dynamic Programming нь дэд асуудлуудын давхардлыг шийдэхэд үр дүнтэй, Greedy нь илүү энгийн бөгөөд шууд шийдэл шаарддаг. Жишээ: Fractional Knapsack-д Greedy ашиглаж болох боловч 0/1 Knapsack-д заавал Dynamic Programming хэрэгтэй.

Ашигласан эх сурвалж:

<https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/>

<https://www.w3schools.com/>

<https://www.programiz.com/>