**ШИНЖЛЭХ УХААН ТЕХНОЛОГИЙН ИХ СУРГУУЛЬ**
**Мэдээлэл холбооны технологийн сургууль**


# БИЕ ДААЛТЫН ТАЙЛАН

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

**Шалгасан багш:**

**Бие даалтын ажил гүйцэтгэсэн:**
B221910027 А. Билгүүн

**Улаанбаатар хот**
**2024**

## Divide-and-Conquer

Divide-and-Conquer нь олон салаатай рекурс дээр суурилсан алгоритм юм. Асуудлыг дэд бодлогод хувааж, тус бүрийг рекурсив байдлаар шийддэг ба дэд асуудлын шийдлүүдийг нэгтгэж анхны асуудлыг шийддэг. Энэ арга нь ижил асуудлын жижиг тохиолдлуудад хуваагдаж, дэд асуудлын шийдлүүдийг нэгтгэж ерөнхий асуудлыг шийдвэрлэх боломжтой асуудлуудад хэрэгтэй.

Алхмууд:

- Хуваах: Асуудлыг ихэвчлэн ижил хэмжээтэй жижиг дэд асуудалд хуваана.
- Conquer: Дэд асуудал бүрийг рекурсив байдлаар шийднэ.
- Нэгтгэх: Анхны асуудлыг шийдэхийн тулд дэд асуудлын шийдлүүдийг нэгтгэнэ.

Merge-sort бол “Divide-and-Conquer” аргын сонгодог жишээ юм. Зорилго нь массивыг эрэмбэлэх. Энэ асуудлыг яаж хэсэгчлэх аргаар шийдэхийг жишээгээр харцгаая.

Жишээ нь бидэнд [38, 27, 43, 3, 9, 82, 10] ийм массив өгөгдсөн гэж үзье.
Алхам 1(Хуваах): Массивыг хоёр хэсэг болгон хуваана.

- [38, 27, 43, 3] болон [9,  82, 10] гэж хуваагдана.

Алхам 2 (Conquer): Хоёр талыг давтах байдлаар ангил.

- Зүүн тал [38, 27, 43, 3] нь [38, 27] ба [43, 3] гэж хуваагдана. Эдгээр нь рекурсив байдлаар хуваагдаж ангилагдаад [27, 38] ба [3, 43] болно.
- Баруун тал [9, 82, 10] нь мөн адил хуваагдаж, [9] ба [10, 82] гэж ангилагдана.

.Алхам 3 (Нэгтгэх): Эрэмбэлэгдсэн талыг нэгтгэнэ.

- [27, 38] ба [3, 43]-ийг [3, 27, 38, 43].  [9] ба [10, 82]-ийг [9, 10, 82] болгон нэгтгэнэ.
- Эцэст нь эрэмбэлэгдсэн хоёр талыг [3, 27, 38, 43] ба [9, 10, 82]-ийг  [3, 9, 10, 27, 38, 43, 82] болгон нэгтгэнэ.

```
function mergeSort(array):
   if length(array) <= 1:
      return array
   mid = length(array) / 2
   leftHalf = mergeSort(array[0:mid])
   rightHalf = mergeSort(array[mid:length(array)])
   return merge(leftHalf, rightHalf)

function merge(left, right):
   result = []
   while left is not empty and right is not empty:
      if left[0] <= right[0]:
         result.append(left[0])
         left.remove(left[0])
      else:
         result.append(right[0])
         right.remove(right[0])

   while left is not empty:
      result.append(left[0])
      left.remove(left[0])

   while right is not empty:
      result.append(right[0])
      right.remove(right[0])

   return result
```

# Dynamic Programming (DP)

Динамик программчлал нь давхцаж буй дэд асуудал, оновчтой дэд бүтэцтэй асуудлыг шийдвэрлэхэд ашигладаг оновчлолын арга юм. Нэг дэд асуудлын шийдлүүдийг дахин дахин тооцоолохын оронд дэд асуудал бүрийг нэг удаа шийдэж, үр дүнг ирээдүйд ашиглах зорилгоор хүснэгтэд хадгалдаг.

Жишээ бодлого: Фибоначчийн дараалал

> Фибоначчийн дэс дараалал рекурсив хэрэгжилтийн хувьд $F(n)$ хандах бүрд $F(n-1)$ болон $F(n-2)$ утгуудыг олон удаа дахин тооцоолохыг хамардаг бөгөөд энэ нь экспоненциал хугацааны нарийн төвөгтэй байдалд хүргэдэг.

Динамик программчлалын шийдэл:

- Рекурсив хамаарал: $F(n)=F(n-1)+F(n-2)$, $F(0)=0$ ба $F(1)=1$.
- Фибоначчийн тоонуудыг массив дотор хадгалж, шаардлагатай үед хадгалсан утгуудыг дахин ашиглана.

Жишээ нь: $F(2)$ ийг тооцоолохын тулд өмнөх хоёр хадгалагдсан утга болох $F(0)=0$ ба $F(1)=1$ хоёрыг ашиглаж тооцно. $F(2) = F(1) + F(0) = 3$.

Ажиллах хурдны хувьд фибоначчийн тоо бүрийг зөвхөн нэг удаа тооцоолж, ирээдүйд ашиглахаар хадгалдаг тул цаг хугацааны нарийн төвөгтэй байдал $O(n)$ болж буурдаг.

```
function fibonacci(n):
   if n <= 1:
      return n

   dp = array of size (n + 1) initialized to 0
   dp[0] = 0
   dp[1] = 1

   for i from 2 to n:
      dp[i] = dp[i - 1] + dp[i - 2]

   return dp[n]
```

# Greedy Algorithms

Хомхойлох алгоритм нь оновчтой шийдлийг олохын тулд алхам бүрд хамгийн оновчтой сонголтыг хийдэг оновчлолын алгоритмын нэг төрөл юм.Энэ нь урт хугацааны үр дагаврыг тооцохгүйгээр "хамгийн сайн хувилбарыг одоо авах" зарчмаар ажилладаг.

Жишээ бодлого: Fractional Knapsack Problem
Тус бүр нь жинтэй, үнэ цэнтэй зүйлсийн багцыг өгснөөр тэдгээрийг тогтмол багтаамжтай цүнхэнд хийснээр олж болох хамгийн их утгыг тодорхойлох зорилготой юм. item-ийг бас хувааж авч болно.

Алхмууд:

- Алхам 1: Item тус бүрийн жингийн үнэ цэний харьцааг (жишээ нь, жингийн нэгжийн үнэ цэнэ) тооцоол.
- Алхам 2: Item-ийг үнэ цэнэ жингийн харьцаагаар нь буурах дарааллаар эрэмбэл.
- Алхам 3: Жингийн хамгийн өндөр үнэ цэний харьцаатай зүйлсийг сонгоод үүргэвчийг дүүртэл нь хийнэ. Хэрэв дараагийн item бүрэн багтахгүй бол тохирох хэсгийг нь авна.

Items:
Item 1: Үнэ = 60, Жин = 10
Item 2: Үнэ = 100, Жин = 20
Item 3: Үнэ = 120, Жин = 30
Цүнхний багтаамж: 50

Алхам алхмаар шийдэл:

1. Үнэ ба жингийн харьцаа:
    - Item 1: 60 / 10 = 6
    - Item 2: 100 / 20 = 5
    - Item 3: 120 / 30 = 4
2. item-ийг жингийн харьцаагаар эрэмбэлэх:
    - Item 1 (харьцаа = 6)
    - Item 2 (харьцаа = 5)
    - Item 3 (харьцаа = 4)
3. Item сонгох
    - Item 1-ийг бүхлээр нь авна.(Цүнхний үлдсэн хэмжээ: 40, Нийт үнийн дүн: 60)
    - Item 2-ийг бүхлээр нь авна.(Цүнхний үлдсэн хэмжээ: 20, Нийт үнийн дүн: 160)
    - Item 3-ийг 2/3 аар хэсгээр нь авна. Учир нь item 3-ийн нийт хэмжээ 30 цүнхний үлдсэн хэмжээ 20 болохоор. 120 * 2/3 = 80. (Цүнхний үлдсэн хэмжээ: 0, Нийт үнийн дүн: 220)

Хомхойлох алгоритм нь бутархай цүнхний асуудлыг шийдвэрлэхэд тохиромжтой, учир нь та эд зүйлсийн бутархай хэсгийг авахыг зөвшөөрдөг бөгөөд энэ нь алхам бүрд жингийн нэгжийн хамгийн өндөр утгатай зүйлийг сонгох нь ирээдүйд оновчтой шийдлийг баталгаажуулдаг гэсэн үг юм.

## Recursion vs Divide-and-Conquer

Рекурс гэдэг функц нь асуудлыг ижил асуудлын жижиг тохиолдлуудад задлахын тулд өөрийгөө дууддаг ерөнхий программчлалын арга юм. Математикийн дарааллыг тооцоолох, хайлт хийх, өгөгдлийн бүтцээр дамжих зэрэг янз бүрийн ажлуудад ашиглаж болно. Гэсэн хэдий ч рекурс нь дэд асуудлууд нь бие даасан эсвэл давхацдаггүй гэсэн үг биш юм. Жишээлбэл, Фибоначчийн тоог рекурс ашиглан тооцоолохдоо энэ функц өөрийгөө $F(n-1)$ болон $F(n-2)$ гэж дахин дахин дууддаг бөгөөд энэ нь тооцоолол давхцаж, үр ашиггүй болоход хүргэдэг.

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

Divide-and-Conquer гэдэг нь асуудлыг жижиг, бие даасан дэд асуудал болгон үр дүнтэй задалдаг рекурсив хандлагын тодорхой төрөл юм. Дэд асуудал бүрийг рекурсив аргаар шийдэж, дараа нь шийдлүүдийг нэгтгэж эцсийн үр дүнг бүрдүүлнэ. Divide-and-Conquer сонгодог жишээ бол Merge Sort алгоритм юм. Merge Sort-д оролтын массивыг хоёр хагаст хувааж, хагас бүрийг рекурсив дуудлагуудыг ашиглан тусад нь эрэмбэлж дараа нь эрэмбэлэгдсэн хоёр талыг буцааж нэгтгэдэг. Энэхүү бүтэцлэгдсэн арга нь Merge Sort-д O(nlogn) цаг хугацааны нарийн төвөгтэй байдлыг бий болгох боломжийг олгодог бөгөөд энэ нь давхцаж буй дэд асуудлуудтай асуудлуудын хувьд гэнэн рекурсив шийдлүүдээс илүү үр дүнтэй байдаг.

```
function fibonacci(n):
    if n <= 1:
        return n

    mid = n // 2
    a = fibonacci(mid)
    b = fibonacci(mid + 1)

    if n % 2 == 0:
        return a * (2 * b - a)  // F(2k) = F(k) * (2*F(k+1) - F(k))
    else:
        return b * b + a * a    // F(2k + 1) = F(k+1)^2 + F(k)^2
```

Тайлбар:

1. Үндсэн тохиолдол: Функц нь n нь 0 эсвэл 1 эсэхийг шалгадаг. Хэрэв тийм бол шууд буцаана.
2. Хуваах: Уг функц нь асуудлыг хоёр дэд асуудалд хуваахын тулд голын индексийг тооцоолно (mid = n // 2) .
    - F(k)-ийг илэрхийлэх фибоначчи(mid)-ийг тооцоол.
    - F(k+1)-ийг илэрхийлэх фибоначчи(mid +1)-ийг тооцоол.
3. Нэгтгэх: Хоёр дэд бодлогын үр дүнг n нь тэгш эсвэл сондгой эсэхээс хамаарч нэгтгэнэ.
    - Хэрэв n нь тэгш бол Фибоначчийн тоог $F(2k)=F(k)\times(2\times F(k+1)-F(k))$ томьёогоор тооцоолно.
    - Хэрэв n нь сондгой бол Фибоначчийн тоог $F(2k+1)=F(k+1)^2+F(k)^2$

## Divide-and-Conquer vs. Dynamic Programming

Divide-and-Conquer, Динамик программчлал (DP) хоёулаа нарийн төвөгтэй асуудлуудыг жижиг дэд асуудал болгон хувааж шийдвэрлэх стратеги юм. Үндсэн ялгаа нь эдгээр дэд асуудлуудыг хэрхэн шийдвэрлэхэд оршдог. "Divide-and-Conquer" нь асуудлыг бие даасан дэд бодлогод хувааж, дэд бодлого бүрийг тусад нь шийдэж, үр дүнг нэгтгэн эцсийн шийдлийг гаргадаг. Энэ нь дэд асуудлууд давхцахгүй, бие даан шийдвэрлэх боломжтой үед үр дүнтэй байдаг. Жишээлбэл, Түргэн эрэмбэлэх алгоритм нь тэнхлэгийн элементийг сонгох, массивыг бага ба түүнээс их элемент болгон хуваах, хуваалтуудыг рекурсив байдлаар эрэмбэлэх замаар хуваах ба байлдан дагуулах аргыг ашигладаг.

Үүний эсрэгээр, Динамик программчлал нь тооцооллын явцад ижил дэд асуудлуудыг хэд хэдэн удаа шийддэг давхцаж буй дэд асуудлуудтай асуудлуудад ялангуяа ашигтай байдаг. АН-д нэмэлт тооцоо хийхээс зайлсхийхийн тулд дэд асуудал бүрийг нэг удаа шийдэж, хүснэгтэд (санах) хадгалдаг. Динамик программчлалын тод жишээ бол 0/1 үүргэвчний асуудал бөгөөд өгөгдсөн жингийн багтаамжтай цүнхэнд багтах зүйлсийн нийт үнэ цэнийг нэмэгдүүлэх зорилготой юм. DP нь зүйлсийг оруулах, хасахтай холбоотой жижиг дэд асуудлуудыг шийдэж, өмнө нь тооцоолсон үр дүнг дахин ашиглах замаар оновчтой утгыг үр ашигтайгаар тооцдог.

```
function fibonacci(n):
   if n <= 1:
      return n

   dp = array of size (n + 1) initialized to 0
   dp[0] = 0
   dp[1] = 1

   for i from 2 to n:
      dp[i] = dp[i - 1] + dp[i - 2]

   return dp[n]
```

Тайлбар:

1. Динамик программчлал:
    - Fibonacci функц нь мөн n нь 1-ээс бага эсвэл тэнцүү эсэхийг шалгаж, үндсэн тохиолдолд n-ийг буцаана.
    - Fibonacci утгыг хадгалахын тулд dp массивыг эхлүүлж, эхний хоёр утгыг тодорхой зааж өгдөг.
    - $F(n)=F(n-1)+F(n-2)$ хамаарал дээр үндэслэн массивыг дүүргэж, 2 -> n хүртэл давтагдана.
    - Эцэст нь энэ нь dp[n]-д хадгалагдсан утгыг буцаадаг бөгөөд энэ нь n-р Фибоначчийн тоо юм.

Хоёр арга хоёулаа Фибоначчийн дарааллыг үр дүнтэй тооцоолох боловч Динамик программчлалын арга нь хадгалагдсан үр дүнг ашиглах зэргээс шалтгаалан n-ийн том утгуудад илүү үр дүнтэй байдаг. Divide-and-Conquer арга нь рекурсын хувьд илүү гоёмсог боловч хадгалахгүйгээр илүүдэл тооцооллоос болж удаашралтай байдаг.

## Greedy vs. Dynamic Programming

Хомхойлох алгоритмууд болон Динамик программчлал нь шийдлүүдийг оновчтой болгохыг эрэлхийлдэг боловч шийдвэр гаргах арга барилаараа эрс ялгаатай байдаг. Хомхойлох алгоритмын сонголтууд нь ирээдүйн хэмжээнд оновчтой шийдэлд хүргэнэ гэж найдаж тухайн үедээ оновчтой сонголтыг хийдэг. Энэ арга нь "шуналтай сонголтын шинж чанар" харуулсан асуудлуудад сайн ажилладаг бөгөөд алхам бүрд хамгийн сайн сонголтыг сонгох нь ерөнхий оновчтой үр дүнд хүргэдэг. Жишээлбэл, эхлэх болон дуусах хугацаатай үйл ажиллагааны багцыг давхцалгүйгээр сонгох ёстой "Үйл ажиллагаа сонгох" асуудалд шунахай алгоритм нь үйл ажиллагааг дуусгах хугацаанд нь ангилж, алхам бүр дээр хамгийн эрт дуусгах үйл ажиллагааг сонгодог. Энэ арга нь зөрчилдөөнгүйгээр оролцох боломжтой хамгийн олон тооны үйл ажиллагааг үр дүнтэй олдог.

```
function activitySelectionGreedy(activities):
   # Алхам 1: Үйл ажиллагааг дуусах хугацаанд нь ангилах.
   sort(activities by finish time)

   # Алхам 2: Сонгосон үйл ажиллагааг явуулах жагсаалтыг эхлүүлэх
   selected = []

   # Алхам 3: Эхний үйл ажиллагааг сонгоно.
   selected.append(activities[0])

   # Хамгийн сүүлд сонгосон үйл ажиллагааны дуусах хугацааг хянах.
   lastFinishTime = activities[0].finish

  # Алхам 4: Үлдсэн үйлдлүүдийг давтна.
   for i from 1 to length(activities) - 1:
      if activities[i].start >= lastFinishTime:
         selected.append(activities[i])  # Үйл ажиллагааг сонгоно.
         lastFinishTime = activities[i].finish  # Сүүлийн дуусах хугацааг шинэчилнэ.

   return selected  # Сонгогдсон үйл ажиллагааны жагсаалтыг буцаана.
```

Үүний эсрэгээр, Динамик программчлал нь холбогдох бүх дэд асуудлыг шийдэж, тэдгээрийн үр дүнг ашиглан эцсийн хариултыг бий болгох замаар оновчтой шийдлийг баталгаажуулдаг. DP нь давхцаж буй дэд асуудлууд, оновчтой дэд бүтэцтэй асуудлуудад үр дүнтэй байдаг. Үүний нэг жишээ бол өгөгдсөн хоёр дарааллаар ижил дарааллаар гарч ирэх хамгийн урт дарааллыг олох зорилготой хамгийн урт нийтлэг дэд дараалал (LCS) бодлого юм. Динамик программчлалыг ашигласнаар нэмэлт тооцооллоос зайлсхийхийн тулд дэд асуудал бүрийг хүснэгтэд хадгалдаг бөгөөд энэ нь алгоритмд өмнө нь тооцоолсон утгууд дээр үндэслэн шийдлийг үр дүнтэй бүтээх боломжийг олгодог.

Хомхойлох алгоритмууд нь ихэвчлэн энгийн хэрэгжилттэй илүү хурдан шийдлүүдийг санал болгодог ч илүү төвөгтэй хувилбаруудад оновчтой үр дүнд хүрч чадахгүй байж магадгүй юм. Үүний эсрэгээр, Динамик программчлал нь ихэвчлэн тооцоолоход илүү эрчимтэй боловч асуудлын тал бүрийг цогцоор нь авч үзэх замаар оновчтой байдлыг хангадаг.

**Sources:**

1.Ном

- **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein**. *Introduction to Algorithms*. 3rd ed., MIT Press, 2009.

2.Online Resources

- **GeeksforGeeks**."Greedy Algorithm." https://www.geeksforgeeks.org/greedy-algorithms/
- **GeeksforGeeks**."Dynamic Programming.": https://www.geeksforgeeks.org/dynamic-programming/
- **MIT OpenCourseWare**. *Introduction to Algorithms (6.006)*. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/