ДАА Семинар 13

Динамично програмиране

- Алчните алгоритми, избиращи на всяка итерация най-добър ход, не винаги достигат до глобално оптимално решение
  ↳ // пример: Задачата за раницата 0/1 - При дадена раница, нейния максимален капацитет в килограми и N артикула, всеки със своята цена и килограми да се намери максималната стойност, която раницата може да побере.
  Нека капацитетът е 50кг и има три артикула:
  - A_1: цена = 60, тегло = 10
  - A_2: цена = 100, тегло = 20
  - A_3: цена = 120, тегло = 30

  ↳ ако критерият, по който алчният алгоритъм ще избира текущ артикул е максимална стойност $\frac{цена}{тегло}$, той първо ще сложи A_1 ($\frac{60}{10} = 6$), после A_2 ($\frac{100}{20} = 5$) и няма да има място за A_3. Ще изкара резултат 160, но ако сложим първо A_2 и A_3, ще получим по-добро решение - 220, което е и оптимално. Алчният подход не върши работа.

- Изчерпващите алгоритми, проверяващи всяко едно възможно решение, винаги изкарват глобално оптимално решение, но са скъпи откъм време
  ↳ // пример: Пак в задачата за раницата при N = 100 и капацитет 1000 всички възможни решения са 2^{100}, което не може да бъде изчислено за смислено време

- Динамичното програмиране съчетава плюсовете на горните два подхода - дава възможност за дизайн на алгоритми, които последователно генерират вс. възможни решения (гарантирайки коректност), запазвайки вече получените резултати (на подзадачите) (гарантирайки бързина).
  ↳ техника за ефикасна имплементация чрез запазване на вече пресметнати резултати (наивните алгоритми преизчисляват едни и същи подпроблеми отново и отново)

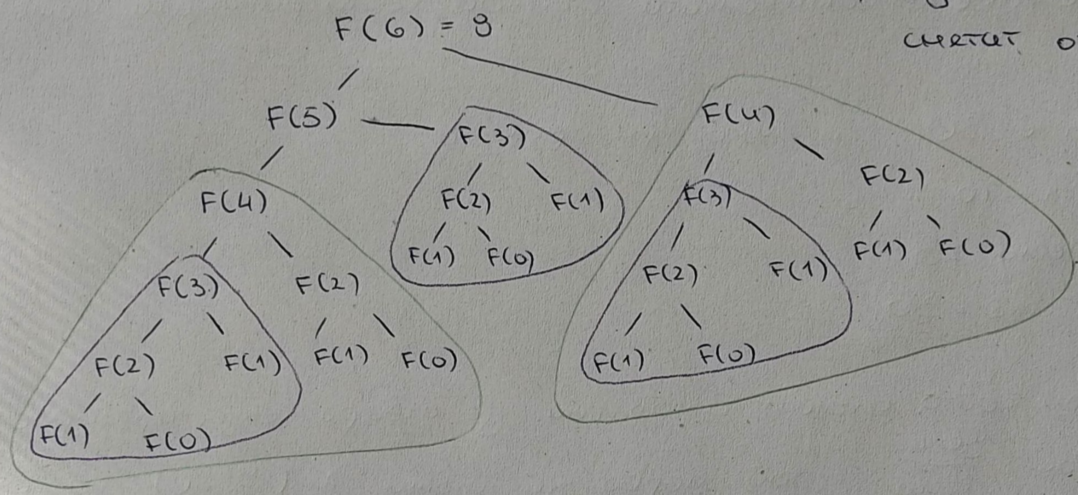
- ДП е компромис с паметта за сметка на времето

// пример: N-тото число на фибоначи

Наивна имплементация:

```
Fib(N)
if N≤1
    return N
return Fib(N-1) + Fib(N-2);
```

↳ дървото на рекурсията при N=6:



→ едни и същи стойности се смятат отново и отново

Чрез динамично:

```
FibDP(N)
Res[1..N]          //може да се
Res[0] ← 0           пазят само
Res[1] ← 1           последните две
                     стойности
for i ← 2 to N
    Res[i] = Res[i-1] + Res[i-2]
return Res[N]
```

При динамичното програмиране посоката на изчисление е отдолу нагоре ↳ директно първо пресметнахме „малките" стойности на Fib, за да получим Fib(N).

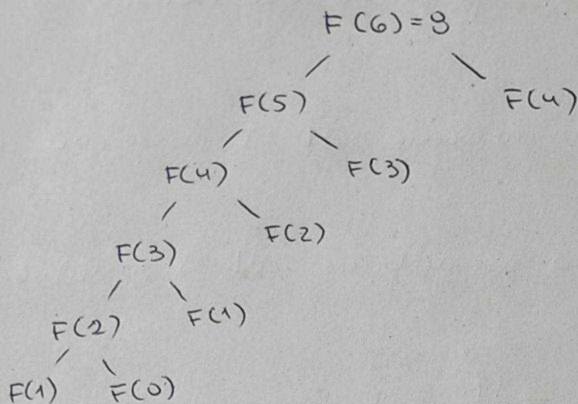
Чрез мемоизация:

```
FibMemo(N, Res)
if Res[N] = -1
    Res[N] = FibMemo(N-1, Res) + FibMemo(N-2, Res)
return Res[N]
```

```
FibInit(N)
Res[1..N]
Res[0] ← 0
Res[1] ← 1
for i ← 2 to n
    Res[i] ← -1
return FibMemo(N, Res)
```

→ само „левите" извиквания се изчисляват, десните ползват вече готовите резултати

→ дървото на рекурсията чрез мемоизация



При мемоизацията има „ход надолу" и „ход нагоре" - същинска рекурсия с пазене на резултатите от предишни викания.

↳ може да се използва и в други рекурсивни алгоритми, в които задачите имат общи подзадачи (няма смисъл да хабим памет и да пазим нещо, което няма да използваме) и компромисът с паметта асимптотично ще подобри времето за изпълнение. (при фиб.: от експоненциално $\xrightarrow[\text{памет}]{\text{доп.}}$ към линейно)

- При дизайн на алгоритми по ДП схема се следват следните стъпки:
  1) Характеристика на оптимално решение
  2) Рекурсивна дефиниция на оптимално решение
  3) Изчисление на решението „от долу нагоре"

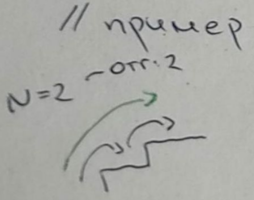
зад ① Дадено е стълбище, състоящо се от N стъпала. В ~~началото~~
Да се изчисли броят на различните начини, по които стълбището може да се изкачи (да се достигне/стъпи на Nтото стъпало), ако може да се минават една или две стълби наведнъж.

а) чрез ДП

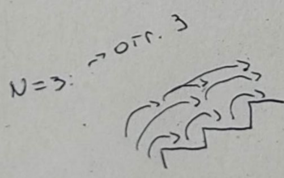
б) чрез мемоизация

// пример

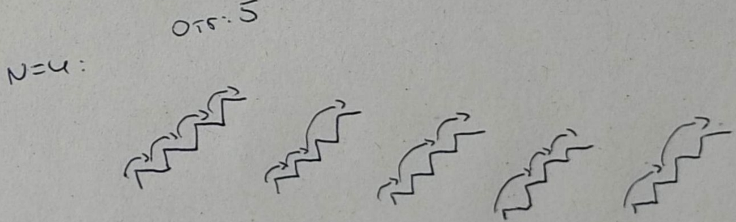
N=2



N=3:



N=4:



Решение:

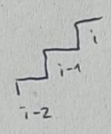
а) Рекурсивна дефиниция:

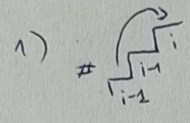
$$d[i] = \begin{cases} 1, & i = 1 \\ 2, & i = 2 \\ d[i-1] + d[i-2] \end{cases}$$

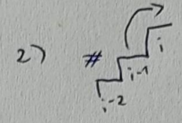
- разбиваме множеството от решения на два дяла - ако знаем броя за d[i-2] и d[i-1], то може да генерираме още различни изкачвания до i-тата стъпка (стъпало/) по следния начин:

1) ако вземем стъпала i-1 и i- наведнъж получаваме d[i-2] различни изкачвания до i-тата стълба/стъпало

2) ако вземем наведнъж само i-тото стъпало получаваме d[i-1] различни изкачвания до i-тото стъпало








```
DPStairs(N)
1. D[1...N]
2. D[1] <- 1
3. D[2] <- 2    // N >= 2
4. for i <- 3 to N
5.      D[i] <- D[i-2] + D[i-1]
6. return D[N]
```

с константна памет

```
DPConstMem(N)
1. back1 <- 1
2. back2 <- 2
3. next <- 0
4. if (n <= 1)
5.      return back1
6. for i <- 3 to N
7.      next <- back1 + back2
8.      back1 <- back2
9.      back2 <- next
10. return back2
```

б) MemoizationStairs(N)

```
1. D[1...N]
2. d[1] <- 1
3. d[2] <- 2
4. for i <- 3 to N
5.      d[i] = -1
6. return StairsRec(N, D)
```

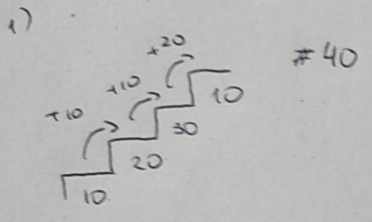
```
StairsRec(N, D)
1. if D[N] != -1
2.      return D[N]
3. D[N] <- StairsRec(N-1, D) + StairsRec(N-2, D)
4. return D[N]
```

Зад ② Дадено е стълбище от N стъпала и жаба, седяща в основата му - първото стъпало.
Жабата иска да стигне до N-тото стъпало. Даден е и масив от височини - HEIGHT, като i-тият му елемент представлява височината на i-тото стъпало.
Жабата може да прескача наведнъж едно или две стъпала, като, ако се намира на i-тото и скочи на j-тото (i+1 или i+2), енергията, която ѝ костваа, е равна на |HEIGHT[i] - HEIGHT[j]|. Да се намери минималното количество енергия, което ще е нужно на жабата, за да стигне до N-тото стъпало.

а) чрез мемоизация
б) чрез ДП

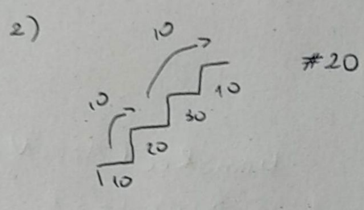
//пример N=4
HEIGHT=[10,20,30,10]

1)



#40

2)



#20

...

Решение:

// защо алчна стратегия не работи? Контрапример: N=6

HEIGHT=[30, 10, 60, 10, 60, 50]

чрез алчна стр.: 60
опт: 40

30 0 10

Рекурсивна дефиниция:

$$D[i] := \begin{cases} 0, & i = 1 \\ |HEIGHT[2] - HEIGHT[1]|, & i = 2 \\ \min(D[i-1] + |HEIGHT[i] - HEIGHT[i-1]|, D[i-2] + |HEIGHT[i] - HEIGHT[i-2]|) \end{cases}$$

а) MemoFrog(N, HEIGHTS)

```
1. D[1..N]
2. D[1] ← 0
3. D[2] ← |HEIGHT[2] - HEIGHT[1]|
4. for i ← 3 to N
5.     D[i] ← -1
6. return MemoRec(N, HEIGHTS, D)
```

може и с константна памет

```
MemoRec(N, HEIGHTS, D)
1. if D[N] != -1
2.     return D[N]
3. D[N] = min( MemoRec(N-1, HEIGHTS, D) +
4.             |HEIGHT[N] - HEIGHT[N-1]|,
5.             MemoRec(N-2, HEIGHTS, D) +
6.             |HEIGHT[N] - HEIGHT[N-2]|)
7. return D[N]
```

б) DPFrog(N, HEIGHTS)

```
1. D[1..N]
2. D[1] ← 0
3. D[2] ← |HEIGHT[2] - HEIGHT[1]|
4. for i ← 3 to N
5.     D[i] ← min(D[i-1] + |H[i] - H[i-1]|, D[i-2] + |H[i] - H[i-2]|)
6. return D[N]
```

Зад ③ Даден е масив от N числа ≥0. Да се намери максималната сума на подредица, която не съдържа два съседни елементи в оригиналния масив

//пример

1) [1, 2, 4]
↳ #5

2) [2, 1, 4, 9]
↳ #11

Решение:

Рекурсивна дефиниция:

$$D[i] := \begin{cases} Arr[i], & i = 1 \\ \max(Arr[i-1], Arr[i]), & i = 2 \\ \max(Arr[i] + D[i-2], D[i-1]) \end{cases}$$

Чрез мемоизация:

```
MemoSum(Arr, N)
1. D[1...N]
2. D[1] ← Arr[1]
3. D[2] ← max(Arr[1], Arr[2])
4. for i ← 1 to N
5.     D[i] ← -1
6. return MemoRec(Arr, N, D)
```

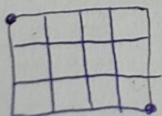
```
MemoRec(Arr, N, D)
1. if D[N] != -1
2.     return D[N]
3. D[N] ← max(Arr[i]+D[i-2], D[i-1])
4. return D[N]
```

Чрез ДП:

```
DPSum(Arr, N)
1. D[1..N]
2. D[1] ← Arr[1]
3. D[2] ← max(Arr[1], Arr[2])
4. for i ← 3 to N
5.     D[i] ← max(Arr[i]+D[i-2], D[i-1])
6. return D[N]
```

Зад ④ Дадена е MxN матрица. Да се намери броят на различните пътища от горния ляв ъгъл до долния десен. Позволени са ходове ↓ и →

//пример:



M = 3
N = 4

Решение:

I.н) От ДС знаем, че всяка различна такава разходка може да се кодира чрез булев вектор (или вектор от „→" и „↓"), като всяка 0-ла представлява ход надолу ↓, а всяка 1-ца - ход надясно →. Трябва да извършим M хода надолу ↓ и N хода надясно →. Търсеният брой е броят на булевите вектори с дължина N+M с точно M нули, този брой е точно

$\binom{N+M}{M}$ (*)

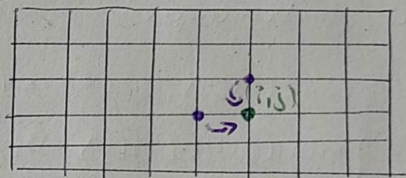
↳

Paths(N, M)
  ... → пресмята (*)

II.н) Чрез мемоизация:

рекурсивна дефиниция:

$$D[i][j] := \begin{cases} 1, & i=1 \ \&\& \ j=1 \\ 0, & i<1 \ || \ j<1 \\ D[i-1][j] + D[i][j-1] \end{cases}$$




```
MemoPaths(N, M)
DP[M][N]
1. DP[1][1] ← 1
2. for i ← 1 to M
3.     for j ← 1 to N
4.         if !(i=1 && j=1)
5.             DP[i][j] = -1
6. return MemoRec(N, M, DP)
```

```
MemoRec(N, M, DP)
1. if i<1 || j<1
2.     return 0
3. if DP[N][M] != -1
4.     return DP[N][M]
5. DP[N][M] = MemoRec(N-1, M, DP) +
6.            MemoRec(N, M-1, DP)
7. return DP[N][M]
```

III.н) Чрез ДП

```
1 DPPaths(N, M)
2 DP[M][N]
3 DP[1][1] ← 1
4 for i ← 1 to M
5     for j ← 1 to N
6         if !(i=1 && j=1)
7.            up ← 0
8.            left ← 0
9.            if i>1  up ← DP[i-1][j]
10            if j>1  left ← DP[i][j-1]
11.           DP[i][j] ← up + lef
12. return DP[N][M]
```

⑤ Дадена е NxM матрица. Всяка клетка в матрицата съдържа някаква цена. Да се намери минималната цена на път от (1,1) до (N,M), позволени са само ↓, → ходове

// пример N=2, M=3

	1	2	3
1	5	9	6
2	11	5	2

цена: 23
цена: 21
...

Решение:

Алчна стратегия не работи. Пример:

N = 3, M = 3

	1	2	3
1	10	8	2
2	10	5	100
3	1	1	2

Ход на алчна стратегия: → → ↓ ↓
Ход на по-добра стратегия ↓ ↓ → →

Рекурсивна дефиниция:

$$D[i][j] := \begin{cases} price[i][j], & i=1 \;\&\&\; j=1 \\ \min(price[i][j] + D[i-1][j],\; price[i][j] + D[i][j-1]) & // i-1 \geq 1 \;\&\; j-1 \geq 1 \end{cases}$$

I) Чрез мемоизация

```
MinCostPath(N, M, price[N][M])
1. D[N][M]
2. D[1][1] ← price[1][1]
3. for i ← 1 to N
4.     for j ← 1 to M
5.         if !(i=1 && j=1)
6.             D[i][j] ← -1
7. return MemoRec(N, M, price, D)
```

```
MemoRec(N, M, price[N][M], D[N][M])
1. if N < 1 || M < 1 return ∞
2. if D[N][M] != -1
3.     return D[N][M]
4. D[N][M] ← min(price[N][M] + MemoRec(
                     N-1, M, price, D),
5.
6.               price[N][M] + MemoRec(
7.                   N, M-1, price, D))
8. return D[N][M]
```

II) Чрез ДП

→ може да се оптимизира сложността по памет

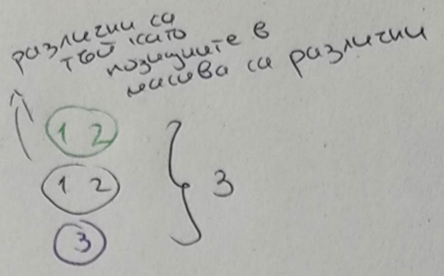
```
MinCostPathDP(N, M, price[N][M])
1. D[N][M]
2. D[1][1] ← price[1][1]
3. for i ← 1 to N
4.     for j ← 1 to M
5.         if !(i=1 && j=1)
6.             up ← ∞
7.             left ← ∞
8.             if i > 1
9.                 up ← D[i-1][j]
10.            if j > 1
11.                left ← D[i][j-1]
12.            D[i][j] ← min(up, left)
13 return D[N][M]
```

зад ⑥ Даден е масив от N положителни числа. Да се изведе броят на различните начини, по които може да изберем елементи от масива, така че сумата на всяка такава колекция да е равна на предварително подадено число Sum.

// пример: N = 4

Sum = 3

Arr [1, 2, 2, 3]



Решение:

Идея: Ако се намираме на позиция i:

1вар) ако елементът на позиция i е ≤ Sum го прибавяме и търсим броя на колекциите в подмасива Arr[1 .. i-1] със сума = Sum - Arr[i]

2вар) не добавяме Arr[i] и търсим броя на колекциите в подмасива Arr[1 .. i-1] със сума Sum

Рекурсивна дефиниция:

$$f(i, Sum) := \begin{cases} 0, & sum = 0 \ \| \ (i = 1 \ \&\& \ Arr[i] > Sum) \\ 1, & i = 1 \ \&\& \ Arr[i] = Sum \\ f(i-1, Sum), & i > 1 \ \&\& \ sum > 0 \ \&\& \ Arr[i] > Sum \\ f(i-1, sum) + f(i-1, Sum - Arr[i]), & \text{иначе} \end{cases}$$

1н) Мемоизация

```
CountMemo(Arr, N, Sum)
1. D[N][Sum]{-1}
2. D[N][0] ← 1 // for i←1 to N
                      D[i][0]←1
3. return MemoRec(Arr, N, Sum, D)
```

```
MemoRec(Arr, N, Sum, D)
1. if D[N][Sum] != -1
2.     return D[N][Sum]
3. if N = 1
4.     return Arr[N] = Sum
5. notTake ← MemoRec(Arr, N-1, Sum, D)
6. take ← 0
7. if Arr[N] <= Sum
8.     take ← MemoRec(Arr, N-1, Sum-Arr[N], D)
9. D[N][Sum] ← notTake + take
10. return D[N][Sum]
```

2н.) ДП

```
CountDP(Arr, N, Sum)
1. D[N][Sum]
2. D[N][0] ← 1
3. if Arr[0] <= Sum
       D[0][Arr[0]] ← 1
4.
5. for i ← 1 to N
6.     for Sumj ← 1 to Sum
7.         notTake ← D[i-1][sumj]
8.         take ← 0
9.         if Arr[i] <= Sumj
10.            take ← D[i-1][Sumj - Arr[i]]
11.        D[i][sumj] ← notTake + take

12. return D[N][Sum]
```