Bit Algo
START

Programowanie dynamiczne

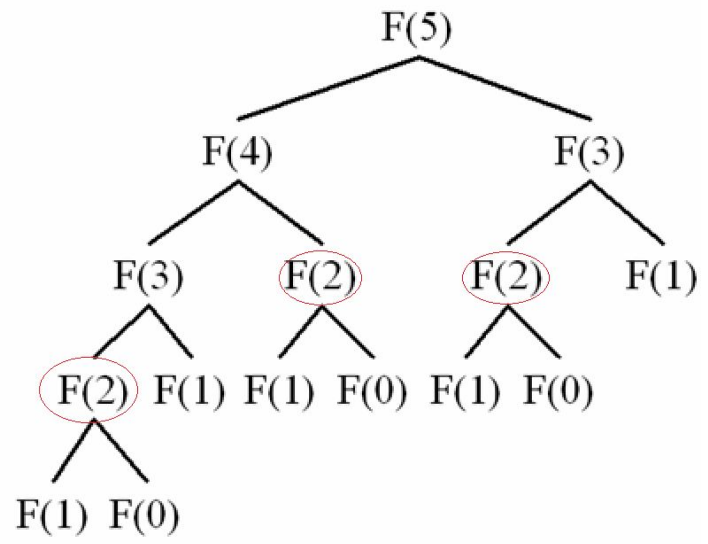
Bit Algo START

Ciąg Fibonacciego rekurencyjnie

$F_0 = 0, F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$

$F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots$



Redundancja! Czy można lepiej (szybciej)?

Programowanie dynamiczne

- **problem:** często w programach rekurencyjnych mamy wielkie drzewo rekurencji, w którym mamy powtarzające się obliczenia
- **idea:** “przycinamy” drzewo - jak raz coś policzymy, to zapamiętujemy wynik i później go wykorzystujemy
- **liczymy daną rzecz tylko raz** - eliminujemy wadę dużych drzew rekurencji, czyli redundantne obliczenia
- **zazwyczaj dobre do problemów optymalizacyjnych** - kiedy pytamy o “najtańszy podział”, “wybór dający największą cenę” etc., to mamy dużą przestrzeń możliwości (możliwe podziały/wybory etc.), którą w naturalny sposób można wyrazić drzewem rekurencyjnym (i je zoptymalizować programowaniem dynamicznym)
- **można realizować na 2 sposoby:** top-down with memoization (dekompozycja), bottom-up (budowanie)

Drobna uwaga językowa

- własność zapamiętywania poprzednio wykonanych obliczeń to **memoization**
- memo - karteczka samoprzylepna
- zapamiętywanie wyników poprzednich obliczeń ~ przyklejanie karteczek z ich wynikami na tablicę
- wykorzystywane w podejściu top-down, bo schodzimy z góry drzewa rekurencji w dół, liczymy i później cofając się po drzewie unikamy dzięki patrzeniu na “karteczki” schodzenia z powrotem w głębokie gałęzie drzewa

Czego wymaga programowanie dynamiczne?

1. **Recursive definition** - musimy być w stanie rekurencyjnie zdefiniować rozwiązanie optymalne -> problem da się rozłożyć na podproblemy
2. **Optimal substructure** - rozwiązanie optymalne da się skonstruować za pomocą optymalnych rozwiązań podproblemów, z których składa się problem -> można łączyć małe optymalizacje w większe i tak aż do całego problemu (tj. “bottom-up”; w drugą stronę - da się zdekomponować, “top-down”)
3. **Overlapping subproblems** - te same podproblemy pojawiają się w wielu gałęziach drzewa rekurencji (poddrzewa “nakładają się” na siebie) -> jak coś raz policzymy, to się później przyda
4. **Independent subproblems** - nieważne, w jakiej kolejności wykonujemy obliczenia podproblemów, zależą one tylko od jeszcze mniejszych podproblemów, z których są składane -> gwarantuje, że podproblemy są identyczne i raz obliczony podproblem możemy wykorzystywać wiele razy bez potrzeby zmian

Uwaga o konstrukcji rozwiązania optymalnego

- w programowaniu dynamicznym bardzo często interesuje nas **wartość** rozwiązania optymalnego, a nie samo rozwiązanie, np. najmniejszy koszt elementów
- czasami interesuje nas też **rozwiązanie optymalne**, np. lista elementów o najmniejszym koszcie
- algorytmy programowania dynamicznego **łatwo dają wartość** (bo muszą ją znać do obliczenia rozwiązania optymalnego)
- zazwyczaj można je zmodyfikować tak, aby obliczając optymalną wartość, **konstruowały rozwiązanie optymalne** i je także zwracały, za cenę skomplikowania algorytmu (i zazwyczaj też pamięci)

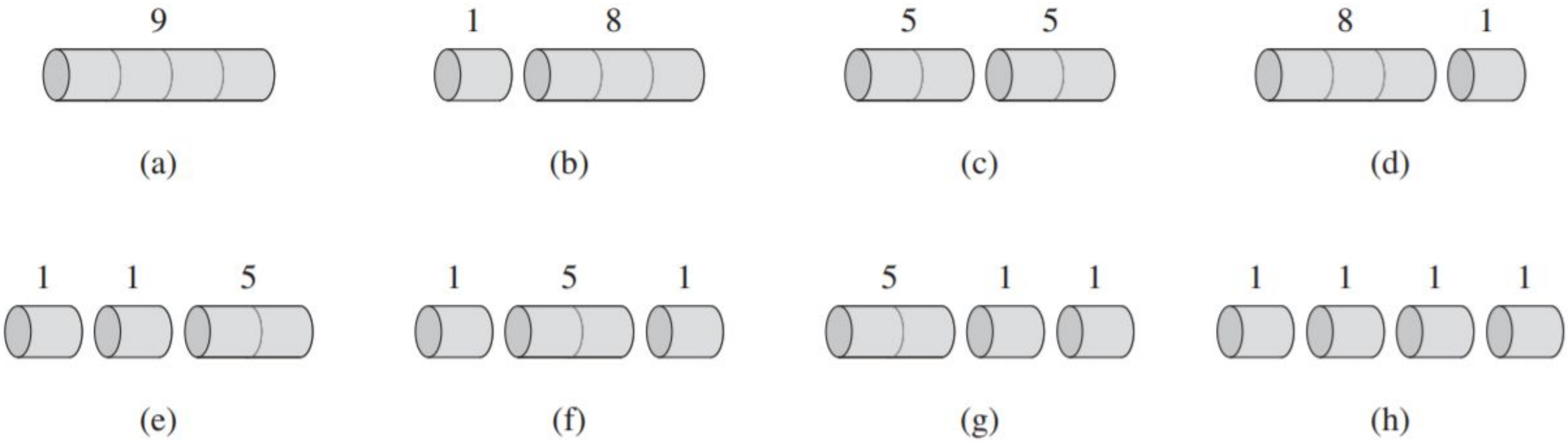
Przykład: cięcie stalowych prętów

Firma kupuje długie stalowe pręty i tnie je na kawałki, które sprzedaje. Kawałki mają długość w metrach wyrażoną zawsze liczbą naturalną. Dla kawałka długości n metrów znane są ceny kawałków długości 1, 2, …, n metrów. Firma chce znać maksymalny zysk, który może uzyskać z pocięcia i sprzedania pręta długości n.

- Dla pręta długości n mamy 2^{n-1} możliwości - eksponencjalna wielkość, brute force nie zadziała
- Szukamy optymalnego rozwiązania problemu

Cięcie stalowych prętów

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Cięcie stalowych prętów

Sprawdzamy warunki programowania dynamicznego:

1. **Recursive definition:** $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$
2. **Optimal substructure:** wynika z powyższego, bo dokonujemy dekompozycji problemu: aby dostać optymalne rozwiązanie problemu dla r_n, wymagamy optymalnych rozwiązań r_{n-1}, r_{n-2} , ... , r_1
3. **Overlapping subproblems:** pojawi się, bo długie kawałki możemy chcieć ciąć wielokrotnie na krótsze kawałki tej samej długości, np. dla cięć kawałków 4 = 2 + 2, 3 = 2 + 1 podproblem dla długości 2 pojawia się wielokrotnie
4. **Independent subproblems:** mamy, bo nieważne, w jaki sposób uzyskaliśmy kawałek danej długości, ważna jest tylko jego długość - cenę sprawdzamy według niej w danej tabeli

Cięcie stalowych prętów - rozw. naiwne

- zwykła rekurencja
- wykorzystuje tylko własność 1. (recursive definition)
- wielokrotnie przelicza w linii 5 te same wartości `p[i]`

```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = −∞
4  for i = 1 to n
5        q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

C. stal. prętów - rozw. top-down with memoization

- tablica pomocnicza do zapamiętywania r[n]
- zewnętrzny wrapper na pomocniczą funkcję rekurencyjną (patrz WDI)
- zaczynamy od największej wartości (tej, która nas interesuje), czyli n

- wartości zapamiętane są większe od 0
- jak mamy zapamiętane - używamy; w przeciwnym wypadku liczymy i zapisujemy
- **nigdy** nie liczymy tego samego dwa razy
- “schodzimy” do 0, a potem “zwijamy” rekurencję

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

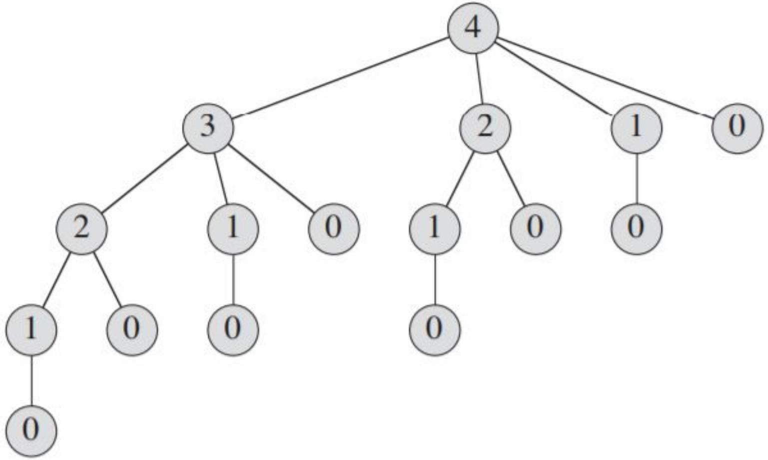
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

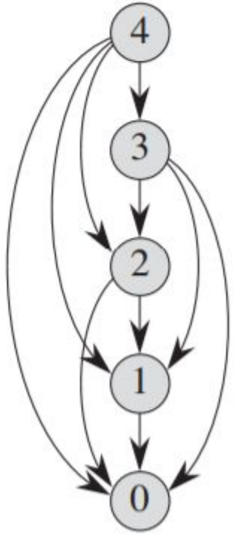
Cięcie stalowych prętów - rozw. bottom-up

- też używamy tablicy pomocniczej
- zaczynamy “od dołu”, najmniejszej wartości, czyli 0
- “budujemy” kolejne, większe rozwiązania
- **gwarancja**, że poprzednie wartości są już w tablicy
- szybsze i oszczędniejsze, bo nie zużywa czasu ani pamięci na rekurencję

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Jaki mamy z tego zysk? n=4





$O(2^n)$

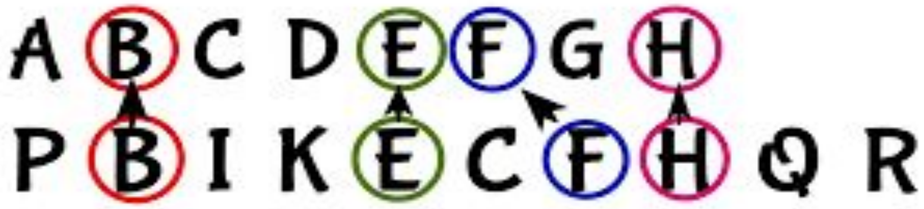
$O(n^2)$

Podsumowanie wstępu teoretycznego

- programowanie dynamiczne nadaje się dobrze do **problemów rekurencyjnych**, **optymalizacyjnych**, pozwala **obliczać optymalne wartości** i ew. **konstruować rozwiązania**
- **idea:** zapamiętywanie poprzednio wykonanych obliczeń, redukcja drzewa rekurencji
- ma warunki: **recursive definition**, **independent and overlapping subproblems**, **optimal substructure**
- 2 podejścia: top-down with **memoization**, bottom-up
- top-down często bardziej intuicyjnie implementuje definicję, bottom-up jest zwykle szybsze i łatwiej konstruować nim rozwiązania

Problem najdłuższego wspólnego podciągu

Cel: Mając na wejściu dwa łańcuchy znakowe (stringi) zwrócić łańcuch znakowy będący najdłuższym wspólnym podciągiem obu łańcuchów wejściowych.
Przykład: dla łańcuchów s1: abdgeffhuj, s2: acdgxffhuy rozwiązaniem jest: adgffhu.



Problem najdłuższego wspólnego podciągu - algorytm naiwny

W sytuacji, w której nie interesuje nas złożoność obliczeniowa algorytmu możemy zastosować proste rozwiązanie rekurencyjne: każdy element (znak) z obu ciągów można po prostu brać, albo nie brać generując w ten sposób wszystkie możliwe podzbiory.

```
LCS (s1, s2, i1, i2):
    if(i1 == length(s1) or i2 == length(s2)):
        return 0
    if s1[i1] == s2[i2]:
        return 1 + LCS(s1, s2, i1+1, i2+1)
    return MAX(
        LCS(s1, s2, i1+1, i2),
        LCS(s1, s2, i1, i2+1)
```

Złożoność O(2^n)

Problem najdłuższego wspólnego podciągu - dynamicznie z zapamiętywaniem

Analizując sygnaturę rekurencyjnej procedury LCS można dojść do wniosku, że liczba unikatowych wywołań (takich mających unikatowy zestaw parametrów) jest zdecydowanie niższa niż 2^n, mianowicie wynosi length(s1)*length(s2). Dlaczego by zatem nie spisywać gdzieś w pamięci już policzonych wywołań?

```
LCS(s1, s2):
        memo = array[length(s1)+1][length(s2)+1]
        return LCSdyn(s1, s2, 0, 0)
```

```
LCSdyn (s1, s2, i1, i2, memo):
        if (memo[i1][i2] is not None):
                return memo[i1][i2]
        result
        if(i1 == length(s1) or i2 == length(s2)):
                result = 0
        if s1[i1] == s2[i2]:
                result = 1 + LCSdyn(s1, s2, i1+1, i2+1, memo)
        result = MAX(
                LCSdyn(s1, s2, i1+1, i2, memo),
                LCSdyn(s1, s2, i1, i2+1, memo))
        memo[i1][i2]= result
        return result
```

Problem najdłuższego wspólnego podciągu - dynamicznie z iteracją

Rozwiązanie na poprzednim slajdzie daje już algorytm wielomianowy. Możliwe jednak, że nie możemy sobie pozwolić na wykorzystanie rekurencji.

```
LCSIteration(s1, s2):
        dp = array[length(s1)+1][length(s2)+1]
        n = length(s1)
        m=length(s2)
        for i = 0 to n:
                dp[i][m] = 0
        for i = 0 to m:
                dp[n][i] = 0
        for i = n-1 to 0:
                for j = m-1 to 0:
                        if s1[i] == s2[j]:
                                dp[i][j] = 1+dp[i+1][j+1]
                        else:
                                dp[i][j] = MAX(dp[i+1][j], dp[i][j+1])
        return dp[0][0]
```

Problem najdłuższego wspólnego podciągu - dynamicznie z iteracją

Rozwiązanie na poprzednim slajdzie daje już algorytm wielomianowy. Możliwe jednak, że nie możemy sobie pozwolić na wykorzystanie rekurencji.

```
LCSIteration(s1, s2):
        dp = array[length(s1)+1][length(s2)+1]
        n = length(s1)
        m=length(s2)
        for i = 0 to n:
                dp[i][m] = 0
        for i = 0 to m:
                dp[n][i] = 0
        for i = n-1 to 0:
                for j = m-1 to 0:
                        if s1[i] == s2[j]:
                                dp[i][j] = 1+dp[i+1][j+1]
                        else:
                                dp[i][j] = MAX(dp[i+1][j], dp[i][j+1])
        return (dp[0][0], dp)
```

No dobra… ale przecież chcemy wiedzieć jak ten podciąg wygląda

```
PrintLCS(s1, s2):
        length, dp = LCSIteration(s1, s2)
        LCS = empty_string(length)
        n= length(s1)
        m=length(s2)
        while(n>0 and m>0):
                if(s1[n-1] == s2[m-1]):
                        LCS[length-1] = s1[n-1]
                        n--, m--,length--
                else if (dp[n-1][m]>arr[n][m-1]):
                        n--
                else:
                        m--
                return LCS
```

X \ Y	0	a 1	s 2	w 3	v 4
0	0	0	0	0	0
a 1	0	↖ **1**	← 1	← 1	← 1
r 2	0	↑ 1	↑ 1	↑ 1	↑ 1
s 3	0	↑ 1	↖ **2**	← 2	← 2
w 4	0	↑ 1	↑ 2	↖ **3**	← 3
q 5	0	↑ 1	↑ 2	↑ 3	← 3
v 6	0	↑ 1	↑ 2	↑ 3	↖ **4**

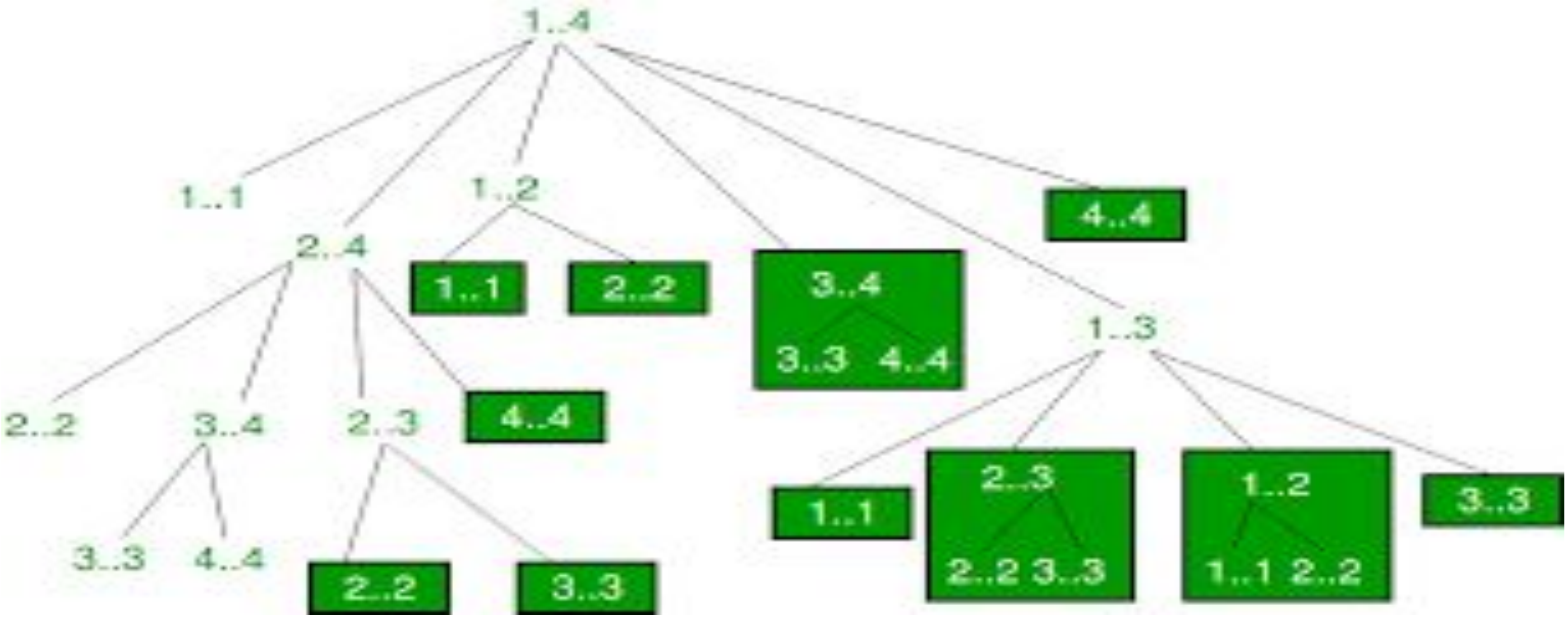
Algorytm za milion dolarów

Cel: mając na wejściu ciąg macierzy, należy podać jak ustawić nawiasy, aby podczas mnożenia ciągu macierzy liczba wykonanych działań arytmetycznych była jak najmniejsza.
Przykład: ciąg trzech macierzy A, B i C o wymiarach odpowiednio 10x30, 30x5, 5x60.
Jeśli mnożenia dokonamy według kolejności (AB)C wykonamy (10*30*5)+(10*5*60) = 4500 operacji.
Jeśli mnożenia dokonamy według kolejności A(BC) wykonamy (30*5*60)+(10*30*60) = 27000 operacji.
Uwaga!: jeżeli mnożymy przez siebie macierze A i B, które mają wymiary odpowiednio n1xm1 i n2xm2 to wykonujemy n1*m1*m2 operacji arytmetycznych.

Mnożenie łańcucha macierzy - teraz próbujemy formalnie.

Krok pierwszy - definicja funkcji dynamicznej (rekurencyjnej), której wartości będziemy zapamiętywać: Niech f(i, j) oznacza minimalną liczbą działań, którą musimy wykonać mnożąc spójny podciąg macierzy o indeksach od i do j. Rozwiązaniem zadania jest wartość funkcji f(0, n-1).
Krok drugi - napisanie rekurencyjnego wzoru na funkcję dynamiczną, tak by wykorzystywała wcześniej obliczone wyniki:
Jeżeli obliczamy f dla podciągu od i do j, to zakładamy, że mamy obliczoną f dla wszystkich wewnętrznych podciągów ciągu od i do j: f(i, j) = MIN[from k = i to j-1] (f(i, k) + f(k+1, j) + rows(i)*columns(k)*columns(j))
Krok trzeci - przypadki brzegowe (warunki końca rekurencji): z definicji f wynika, że nigdy nie podzieli na kawałek mający mniej niż jedną macierz. wtedy wynik jest oczywisty: f(i,i) = 0 dla każdego i.

Mnożenie łańcucha macierzy - graficzna reprezentacja f.



Mnożenie łańcucha macierzy - algorytm.

```
MCM(chain, n):
    dp = array[n][n] // zakładamy, że defaultową wartością jest +infinity
    for i = 0 to n-1: // wypełniamy warunki brzegowe
        dp[i][i] = 0
    for L = 2 to n-1:
        for i = 0 to n-L+1:
            j = i+L-1
            for k =i to j-1:
                dp[i][j] = MIN(dp[i][j],
                        dp[i][k]+dp[k+1][j]+
                        rows(i)*columns(k)*columns(j))
    return dp[0][n-1]
```

O(n^3)

Znowu to samo - chcę jeszcze wiedzieć jak poukładać te nawiasy.

```
MCM(chain, n):
        dp = array[n][n] // zakładamy, że defaultową wartością jest +infinity
        brackets = array[n][n]
        for i = 0 to n-1: // wypełniamy warunki brzegowe
                dp[i][i] = 0
        for L = 2 to n-1:
                for i = 0 to n-L+1:
                        j = i+L-1
                        for k =i to j-1:
                                temp = dp[i][k]+dp[k+1][j]+rows(i)*columns(k)*columns(j)
                                if (temp < dp[i][j]):
                                        dp[i][j] = temp
                                        brackets[i][j] = k
        return dp[0][n-1], brackets
```

Rekurencyjne schody Amazona

Cel: dana jest tablica A zawierająca liczby naturalne nie mniejsze od 1. początkowo stoimy na pozycji 0, wartość A[i]
informuje nas jaka jest maksymalna długość skoku na następną pozycję. Przykład A={1,3,2,1,0}
z pozycji 0 mogę przejść na pozycję 1. z pozycji 1 mogę przejść na 2, 3, 4. Należy policzyć na ile sposobów mogę przejść z pozycji 0 na pozycję n-1, przestrzegając reguł tablicy.

Rekurencyjne schody Amazona - analiza dynamiczna

Krok pierwszy: niech f(i) oznacza na ile sposobów mogę przejść z pozycji i do pozycji n-1. rozwiązaniem zadanie jest oczywiście f(0).
Krok drugi: zauważamy, że od indeksu i wykonuję skok o 1,2,..,A[i], a awrtości f(j) dla j>i mam już policzone. f(i) = sum[from k = 1 to A[i]](f(i+k))
Krok trzeci: jest oczywiste, że f(n-2) = 1.

Rekurencyjne schody amazona - algorytm.

```
Staircase(A, n):
    dp=arra[n-1] // defaultowo mamy tam wartości 0
    dp[n-2]=1
    for i = n-3 to 0:
        for k = 1 to A[i] and i+k<=n-2:
            dp[i]+=dp[i+k]
    return dp[0]
```

O(n*max(A))

Zadania

Zadanie 0

Napisz funkcję obliczającą n-ty wyraz ciągu Fibonacciego z użyciem programowania dynamicznego.

Porównaj jego szybkość działania i złożoność pamięciową z rozwiązaniem iteracyjnym z WDI. Które jest lepsze? Dlaczego?

Zadanie 1

Zmodyfikuj rozwiązanie problemu cięcia stalowych prętów tak, aby konstruowało i zwracało także rozwiązanie, tj. listę długości prętów o największej cenie. Podpowiedź: bottom-up będzie łatwiej

Zadanie 2

Dostajemy liczbę naturalną n. Naszym zadaniem jest policzenie wszystkich binarnych (0/1) string'ów bez jedynek obok siebie

Zadanie 2 - kod

```
def countStrings(n):
    array=[[-1 for i in range(2)] for j in range (n+1)]
    array[0][0], array[0][1], array[1][0], array[1][1] = 0, 0, 1, 1
    for i in range(2,n+1):
        array[i][0] = array[i-1][0] + array[i-1][1]
        array[i][1] = array[i-1][0]
    return array[n][0] + array[n][1]
```

Zadanie 3

Dostajemy listę wartości. Gramy z drugim graczem. Wybieramy zawsze jedną wartość z jednego z końców tablicy i dodajemy do swojej sumy, a następnie to samo robi nasz przeciwnik. Zakładając, że przeciwnik gra optymalnie, jaką maksymalną sumę możemy uzbierać?

“Uogólniony problem paczki mentosów”

Zadanie 4

Dostajemy tablicę (M x N) wypełnioną wartościami(kosztem wejścia). Mamy znaleźć minimalny koszt potrzebny do dostania się z pozycji [0][0] do [M-1][N-1]

Wprowadzimy na początek pewne ułatwienia:

1. Możemy poruszać się tylko w bok i w dół
2. Wszystkie koszty są dodatnie

Zadanie 5

Dostajemy tablicę (M x N) wypełnioną wartościami. Mamy za zadanie znaleźć najdłuższą ścieżkę w tej tablicy (możemy przechodzić na pola sąsiadujące krawędziami), o rosnących wartościach (to znaczy, że z pola o wartości 3, mogę przejść na pola o wartości większej bądź równej 4).

Na początku wprowadzimy ponownie pewne ułatwienie:

1. Mamy dany punkt początkowy

Inne przykłady i zadania

Te i wiele innych przykładów zadań z rozwiązaniami i kodem (w C++) znajdziecie w tym linku

https://blog.usejournal.com/top-50-dynamic-programming-practice-problems-4208fed71aa3

Część z tych zadań wymaga jednak dodatkowych struktur takich jak hashmapy, a część nie wymaga, ale autor i tak z nich korzysta

Bit Algo
START