

Memoisering og dynamisk programmering

Magnus Botnan

botnan at stud.ntnu.no

23/10-2009

Memoisering og DP

Memoisering:

Definition: Save (memoize) a computed answer for possible later reuse, rather than recomputing the answer.

DP:

Definition: Solve an optimization problem by caching subproblem solutions (memoization) rather than recomputing them.

Kilde: <http://www.itl.nist.gov/div897/sqg/dads/HTML/dynamicprog.html>

M

DP

Memorization:

Definition: Save (memoize) a computed answer so that it can be later reuse, rather than recomputing the answer.

DP:

Definition: Solve an optimization problem by combining subproblem solutions (memoization) rather than recomputing.

Kilde: <http://www.itl.nist.gov/div897/sqg/aads/HTML/dynamicprog.html>

Memoisering og DP

Eksempler i dag:

- Fibonacci-tall
- Dyreste vei i trekant
- Longest Increasing Subsequence
- Longest Common Subsequence

Fibonacci!

- Problem: Regn ut Fibonacci-tall nummer n
- Matematisk definisjon:
 - $f(0) = 0$
 - $f(1) = 1$
 - $f(n) = f(n - 1) + f(n - 2)$ for $n > 1$
- Hvert tall unntatt de to første er altså summen av de to foregående:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

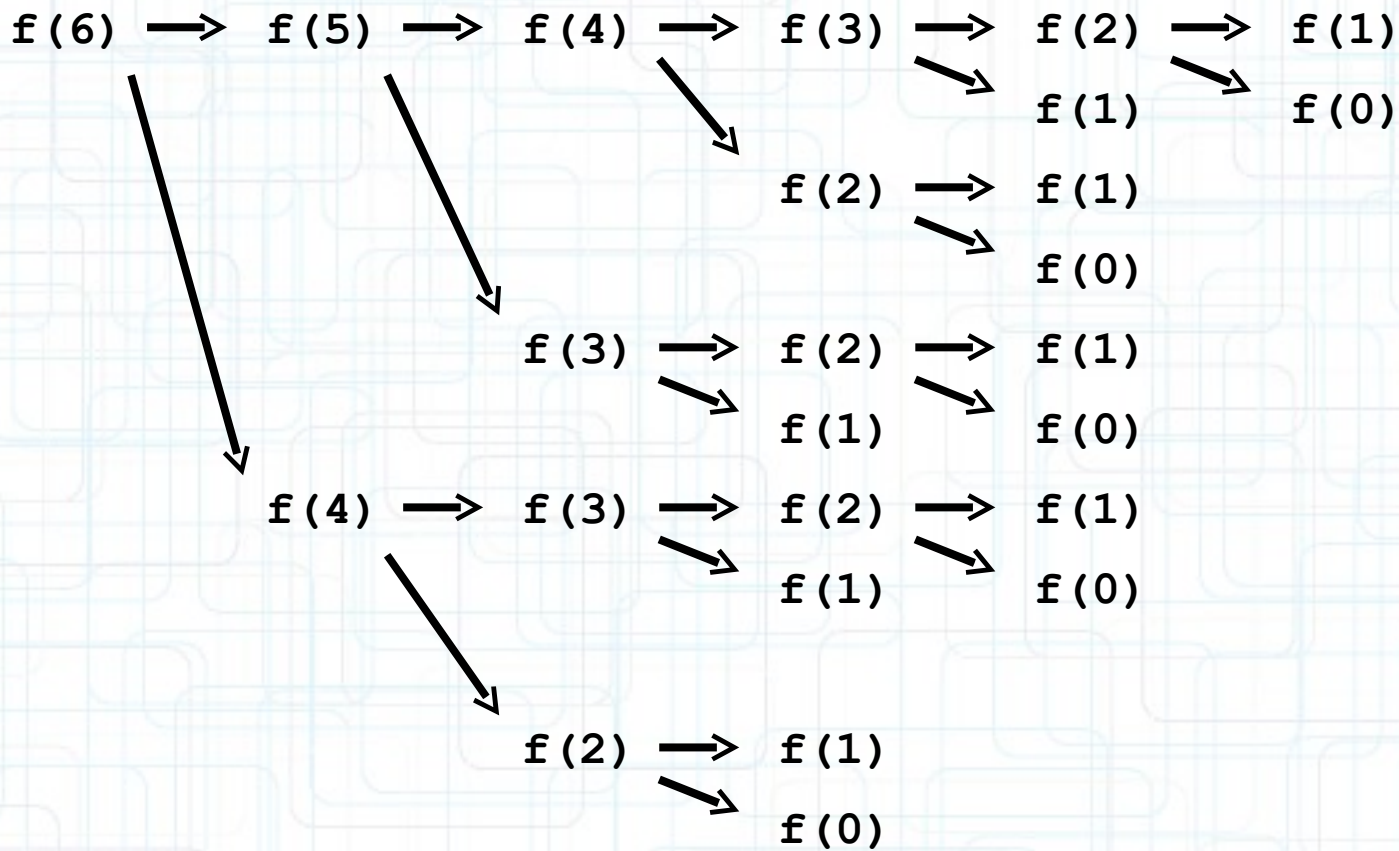
Fibonacci – treg

```
def f(n):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return f(n - 1) + f(n - 2)
```

Fibonacci - Analyse

- Dessverre blir kjøretiden helt forferdelig!
 - For $n < 20$ går det bra
 - Utregning av $f(25)$ tar et halvt sekund
 - $f(30)$ tar fem sekunder
 - $f(35)$ tar et minutt
 - $f(42)$ vil vare ut forelesningen
 - $f(50)$ vil ta et år
- Når n øker med 1, dobles nesten kjøretiden
- Hva skyldes dette?

Ooops ...



Memoisering

- Triks: lagre $f(n)$ etter det er regnet ut!

Når funksjonen blir bedt om å regne ut et Fibonacci-tall, sjekker vi først om vi allerede har regnet det ut

Har vi allerede regnet det ut, returnerer vi resultatet vi har

Ellers starter vi utregningen på vanlig måte

Memoising m/dictionary

```
def f(n, dict):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    elif dict.has_key(n):  
        return dict[n]  
    else:  
        result = f(n - 1, dict) + f(n - 2, dict)  
        dict[n] = result  
        return result  
  
def fib(n):  
    dict = {}  
    return f(n, dict)
```

Dynamisk Programmering

- Hvert tall avhenger kun av de to siste
- Starter i bunn og går oppover
- Ingen rekursjon

Fibonacci - DP

```
def f(n):  
    if n <= 0:  
        return 0  
    array = [-1] * (n + 1)  
    array[0] = 0  
    array[1] = 1  
    for i in xrange(2, n + 1):  
        array[i] = array[i - 1] + array[i - 2]  
    return array[n]
```

Fibonacci - DP

```
def f(n):  
    if n <= 0:  
        return 0  
    array = [-1] * (n + 1)  
    array[0] = 0  
    array[1] = 1  
    for i in xrange(2, n + 1):  
        array[i] = array[i - 1] + array[i - 2]  
    return array[n]
```

DP plassoptimalisering

- Vi ser at utregningen av hvert tall bare avhenger av de to forrige tallene
- Dermed klarer vi oss med bare å lagre tre tall av gangen – sparer minne
- Merk at dette bare er nyttig hvis man kun er interessert i det siste tallet, og ikke vil beholde tallene underveis

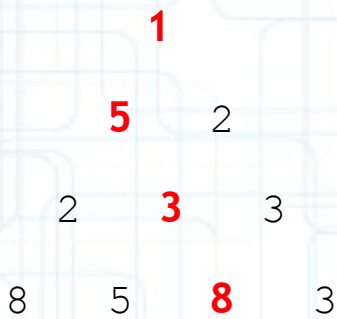
Fibonacci – DP optimal

```
def f(n):  
    if n <= 0:  
        return 0  
    previous = 0  
    current = 1  
    for i in xrange(n - 1):  
        next = previous + current  
        previous = current  
        current = next  
    return current
```

Rekursiv problemformulering

- Det gjelder å formulere løsningen av et problem som en kombinasjon av løsninger av mindre utgaver av samme problem
- Eksempel: Fibonacci-tall nummer n er summen av Fibonacci-tall nummer $n - 1$ og Fibonacci-tall nummer $n - 2$
- De mindre utgavene kalles delproblemer

Dyreste vei i trekant



Dyreste vei i trekant

- Høyde lik 100 gir $2^{99} = 633825300114114700748351602688$ mulige veier!
- 1 milliard veier per sekund gir 20098468420665 år ...

1
5 2
2 3 3
8 5 8 3

DP!

			1	
		5	2	
	2	3	3	
8	5	8	3	

- La $[k,n]$ være element k fra venstre i rad nummer n .

$dyrest[[k,n-1]] = val[k] +$

$\max(dyrest[[k, n]], dyrest[[k+1,n]])$

- Svaret finner vi i $dyrest[[0,0]]$
- Kjøretid $O(n^2)$ (n antall rader)

Memoisering

- Start i toppen og kjør rekursivt
- Lagre etter hvert som det regnes ut
- I nederste rad er dyreste vei i en node bare kostnaden i noden
- Mer jobb ...

Longest Increasing Subsequence (LIS)

- Problem: Gitt en rekke med n reelle tall $a_1, a_2, a_3, \dots, a_n$, finn den største delrekken slik at tallene står i stigende rekkefølge
- Eksempel: [1, 2, 9, 3, 8, 5, 7]
- Fungerer en grådig algoritme?
- Nei, fordi delproblemene ikke er uavhengige. Velger man å ta med et bestemt tall, har man lagt begrensninger på hvilke tall man kan få med seg senere

LIS – Optimal Substruktur

- Vi observerer at en LIS for en bestemt liste består av en LIS for en mindre del av listen
- Eksempel: $[1, 2, 9, 3, 8, 5, 7]$
- LIS'en er $[1, 2, 3, 5, 7]$
- $[1, 2, 3, 5]$ er en LIS for $[1, 2, 9, 3, 8, 5]$
- $[1, 2, 3]$ er en LIS for $[1, 2, 9, 3]$
- $[1, 2]$ er en LIS for $[1, 2]$
- $[1]$ er en LIS for $[1]$

LIS – Optimal Substruktur

Konklusjon: en optimal løsning av problemet består av optimale løsninger av delproblemer

LIS forts.

- La oss se på problemet med å finne en LIS som slutter med tall nummer i , og la oss kalle lengden av dette for $L(i)$
- Da vil hovedproblemet vårt være å finne det beste av $L(1), L(2), L(3), \dots, L(n)$
- Hvordan regner vi ut $L(i)$?

LIS - delproblemer

- Vi finner lengste delsekvensen som ikke inneholder tall større enn tall nummer i

Altså: vi må finne den j som er slik at $j < i$, $a_j < a_i$, og $L(j)$ er størst mulig

$L(i)$ er da lik den største $L(j)$ pluss 1

$$L(i) = \max(L(j) : 0 < j < i \ \&\& \ a_j < a_i) + 1$$

Hvis alle tallene på plasser til venstre for i er mindre enn a_i , er $L(i) = 1$. Dette kan bygges inn i definisjonen over, eller så kan man innføre et ekstra element $a_0 = -\infty$ og definere at $L(0) = 0$

LIS - Tabell

	[1, 2, 9, 3, 8, 5, 7]							
[0, -, -, -, -, -, -, -]								
[0, 1, -, -, -, -, -, -]								
[0, 1, 2, -, -, -, -, -]								
[0, 1, 2, 3, -, -, -, -]								
[0, 1, 2, 3, 3, -, -, -]								
[0, 1, 2, 3, 3, 4, -, -]								
[0, 1, 2, 3, 3, 4, 4, -]								
[0, 1, 2, 3, 3, 4, 4, 5]								

LIS - Sporing

- Nå har vi funnet lengden på LIS'en, men det hadde jo vært kjekt å finne selve LIS'en også...
- I et array P registrerer vi hvilket delproblem hvert delproblem benyttet

a: [1, 2, 9, 3, 8, 5, 7]

L: [0, 1, 2, 3, 3, 4, 4, 5]

P: [-, 0, 1, 2, 2, 4, 4, 6]

LIS - Implementasjon

```
def LIS(a):  
    n = len(a)  
    a.insert(0, -1) # A "fake" element to make  
                    #the real elements start at index 1  
    L = [0] * (n + 1)  
    P = [-1] * (n + 1)  
    for i in range(1, n + 1):  
        bestIndex = 0  
        for j in range(1, i):  
            if a[j] <= a[i] and \  
L[j] > L[bestIndex]:  
                bestIndex = j  
        L[i] = L[bestIndex] + 1  
        P[i] = bestIndex  
    return L, P
```

LIS - Implementasjon

- Koden på forrige lysbilde returnerer både lengde-arrayet L og forgjenger-arrayet P
- Det er ikke sikkert at LIS'en slutter med det siste tallet, så vi må gå gjennom L for å finne den største lengden
- Når vi har gjort det, kan vi spore oss tilbake gjennom P for å finne selve LIS'en
 - Dette kan gjøres iterativt eller rekursivt

LIS - Implementasjon

```
def LIS_result(a, L, P):  
    bestIndex = 1  
    for i in xrange(2, len(L)):  
        if L[i] > L[bestIndex]:  
            bestIndex = i  
    # Use make_LIS_recursively  
    # or make_LIS_iteratively  
    # (from the next slide)  
    return make_LIS(a, P, bestIndex)
```

```
# Example of use  
a = [1, 2, 9, 3, 8, 5, 7, 1]  
L, P = LIS(a)  
print LIS_result(a, L, P)  
# Should print [1, 2, 3, 5, 7]
```

LIS - Implementasjon

```
def make_LIS_recursively(a, P, i):  
    if i == 0:  
        return []  
    else:  
        b = make_LIS_recursively(a, P, P[i])  
        b.append(a[i])  
        return b
```

```
def make_LIS_iteratively(a, P, i):  
    b = []  
    while i != 0:  
        b.append(a[i])  
        i = P[i]  
    b.reverse()  
    return b
```

Longest Common Subsequence

- Finne lengste tegnsekvens som er felles for to strenger
- Eksempel:
A = "abqaeehgpcydkpk1z"
B = "rwazxbaertctdz"
- Vi definerer $LCS(i, j)$ til å være LCS'en til $A[0:i]$ og $B[0:j]$

LCS

- Dersom siste tegn i $A[0:i]$ og $B[0:j]$ er like, er disse en del av $LCS(i, j)$; da trenger vi bare å finne $LCS(i - 1, j - 1)$ og legge til 1
- Hvis ikke, kan ikke begge de siste tegnene være med i $LCS(i, j)$
- Da er $LCS(i, j)$ enten lik $LCS(i - 1, j)$ eller $LCS(i, j - 1)$
- Vi får da en tabell over $LCS(i, j)$ på $m \times n$, hvor m er lengden til A og n er lengden til B

LCS

- $LCS(i,j) =$
 - 0 hvis $i=0$ eller $j=0$
 - $1 + LCS(i-1,j-1)$ hvis $A[i] = B[j]$
 - $\max(LCS(i-1,j), LCS(i,j-1))$ hvis $A[i] \neq B[j]$

LCS - Eksempel

- LCS av AGCAT og GAC

-	Ø	A	G	C	A	T
Ø	0	0	0	0	0	0
G	0	0	1	1	1	1
A	0	1	1	1	2	2
C	0	1	1	2	2	2

Betingelse 1: Optimal Substruktur

- Ofte har vi en situasjon hvor vi ønsker å finne den best mulige løsningen av et problem
- Skal vi kunne bruke DP, må det være slik at problemet har en optimal substruktur – en optimal løsning av hele problemet består av optimale løsninger av delproblemer
- Når vi løser et delproblem, må vi som regel se gjennom alle de mindre delproblemene og velge det beste

Betingelse 1.5: Uavhengige Delproblemer

- Løsningen av ett delproblem må ikke legge begrensninger på hvordan andre delproblemer kan løses – delproblemer må ikke ha felles ressurser som kan spises opp
- (Anti)eksempel: Longest simple path
 - Man skulle tro at en LSP fra a til b må bestå av en LSP fra a til x og fra x til b – men det er ikke sikkert, for hvis disse LSP'ene overlapper, kan de ikke kombineres til en lovlig LSP fra a til b . Vi har altså ikke optimal substruktur, og det skyldes at vi ikke har uavhengige delproblemer – fordi en LSP fra a til x , risikerer vi å "bruke opp" noder som LSP fra x til b trenger.

Betingelse 2: Overlappende delproblemer

- Det burde muligens heller hete "gjentatte delproblemer"
- Løsningen av to forskjellige delproblemer kan involvere løsning av ett eller flere felles delproblemer
- Derfor blir rekursjon uten memoisering ekstremt ineffektivt fordi det samme arbeidet gjøres gang på gang
- Dersom delproblemene ikke overlapper, kan en splitt-og-hersk-algoritme brukes

“Betingelse 3”: Nok plass

- DP: “bytter” tid mot plass.
- Kun brukbart når det er nok plass i minnet til å lagre delproblemene

Grådig og DP

- Begge utnytter optimal substruktur
- Grådig
 - Lokalt optimale valg er globalt optimale; optimal løsning avgjøres ut fra hva som virker best der og da
 - Løses som regel ovenfra og ned; man tar et valg og ender opp med et mindre delproblem
- DP
 - Optimal løsning avgjøres ved å se på optimale løsninger av delproblemer
 - Løses nedenfra og opp; man løser større og større delproblemer ut i fra de mindre delløsningene

DP: Sammendrag

- Rekursiv struktur
- Optimal substruktur
- Overlappende delproblemer
(hvis ikke: splitt og hersk-algoritme)
- Du vet ikke hvilket delproblem du skal benytte før du har løst alle delproblemene
(hvis ikke: grådig algoritme)