



Dynamic Programming

Foundations of Algorithms
Guven

Reading Assignment

- Cormen, Chapter 15.1, 15.3



Dynamic Programming Outline

- Dynamic Programming
- Memoization
- Longest Common Subsequence
- Sequence Alignment



Dynamic Programming

- *"Those who cannot remember the past are condemned to repeat it"*
- Not about writing code
 - The term was coined in 1950s when programming was an esoteric activity
- Dynamic optimization
 - Not greedy
- A general solution method for problems which have the following properties
 - Optimal substructure
 - Overlapping subproblems



Dynamic Programming (DP)

- Divide and conquer \leftrightarrow Hadoop MapReduce
 - Partition problem to disjoint problems
 - Solve them recursively
 - Combine solutions
- Dynamic Programming \leftrightarrow Linux **diff**
 - Subproblems overlap
 - Recursively define the subproblem
 - Find optimal solutions to subproblems
 - Solve the problem using computed optimal solutions



Fibonacci Numbers

- $F_0=0$, $F_1=1$, and $F_n=F_{n-1} + F_{n-2} \forall n \geq 2$
 - $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, .. \rangle$

A recursive solution

```
def fibr(n):  
    if n < 2:  
        return n  
    else:  
        return fibr(n-1)+fibr(n-2)
```

- Time complexity: $O(2^n)$
 - Recurrence: $T(n)=T(n-1)+T(n-2)+O(1)$



Memoization

- Memo(r)ization: Remember everything
 - `fibr(n)` re-computes the same values repeatedly
- A memoized solution

```
F={}
def fibm(n):
    if n<2:
        return n
    elif n not in F:
        F[n] = fibm(n-1)+fibm(n-2)
    return F[n]
```

- Time complexity: $O(n)$



Principle of Optimality

- Independent of the initial state, remaining decisions must be optimal with regard the state following from the first decision
- **Definition:** A problem is said to satisfy the Principle of Optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems
- Necessary condition for dynamic programming



Dynamic Programmed Fibonacci

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

- $O(n)$ time and $O(n)$ space complexity



Improved DP Fibonacci

- Do not maintain the entire $F[i]$

```
def fibdp2(n):  
    prev=1  
    curr=0  
    for i in range(1,n+1):  
        next=curr+prev  
        prev=curr  
        curr=next  
    return curr
```

- $O(n)$ time and $O(1)$ space complexity



Discussion

- The practical time complexity of `fibdp2()` is not $O(n)$
 - $F[n]$ grows exponentially
Storing a particular $F[n]$ requires $\Omega(n)$ time
- We cannot perform arbitrary-precision arithmetic in constant time
- Practically time complexity of improved DP `fibdp2(n)` is $O(n^2)$



Sequence Definitions

- Given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$
 - The sequence $Z = \langle z_1, z_2, \dots, z_k \rangle$ is a **subsequence** of X if there exists strictly increasing sequence $\langle i_1, i_2, \dots, i_k \rangle$ of indices of X such that $\forall j=1, 2, \dots, k$ we have $x_{i_j} = z_j$
- Example
 - $X = \text{ABAZDC}$, $Z = \text{BAC}$ is a subsequence with indices $\langle 2, 3, 6 \rangle$
- Given two sequences X and Y , the sequence Z is a common subsequence of X and Y if Z is a subsequence of both



Longest Increasing Subsequence

- Given a sequence of numbers find the longest increasing subsequence
- Example (consecutive):
 - $\langle 1, 2, 5, 3, 9, 5, 3, 3, 2, 1, 5, 7, 8, 9, 0, 1 \rangle$
 - $\text{LIS} = \langle 1, 5, 7, 8, 9 \rangle$
 - $|\text{LIS}| = 5$
- Example (non-consecutive, permitting insertions)
- $\text{LIS} = \langle 1, 2, 3, 3, 3, 5, 7, 8, 9 \rangle$



Length of LIS

```
def LIS(prev, A[1..n]):  
    # Find the longest consecutive subsequence length  
    # Call with LIS(MIN_INTEGER, A)  
    if n==0:  
        return 0  
    else:  
        max=LIS(prev, A[2..n])    # A[1] not in solution  
        if A[1]>prev:    # Check if A[1] can be added  
            L = 1 + LIS(A[1],A[2..n])  
            if L>max:  
                max = L  
    return max
```



Longest Common Subsequence

- Bioinformatics main application:
 - Comparing two DNA sequences
DNA as a string built from the alphabet {A,C,G,T}
- Example:
 - $X = \text{ABAZDC}$
 - $Y = \text{BACBAD}$
 - Longest Common Subsequence (LCS) = ABAD



Brute Force LCS

- Enumerate all subsequences of X and check each subsequence is also a subsequence of Y
 - Each subsequence of X corresponds to the indices $\langle i_1, i_2, \dots, i_m \rangle$
- Number of possible subsequences:
 - Binary encode the presence of an index, *e.g.* $\langle 0, 1, 1, 0, 0, \dots, 1, 1 \rangle$
 - There are 2^m subsequences \rightarrow exponential



Optimal Substructure of LCS

- “Prefixes”
 - Define the i^{th} prefix of X as $X_i = \langle x_1, x_2, \dots, x_i \rangle$
 - X_0 is empty sequence
- Let $X = \langle x_1, x_2, \dots, x_m \rangle$, $Y = \langle y_1, y_2, \dots, y_n \rangle$ and let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y
 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y
 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}



LCS DP Implementation

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- $b[.]$ is used to store the LCS using SKIPX (\leftarrow), SKIPY (\uparrow) and ADDXY (\nwarrow)
 - Also called backtracking
- Boundary conditions are marked with SKIPX (\leftarrow) and SKIPY (\uparrow)
- $c[.]$ is used for memoization



Implementation

```
def LCS(x[1..m], y[1..n]):  
    c = Matrix(m,n)  
    for i in range(m+1):  
        c[i,0]=0; b[i,0]=SKIPX  
    for j in range(n+1):  
        c[0,j]=0; b[0,j]=SKIPY  
    for i in range(1,m+1):  
        for j in range(1,n+1):  
            if x[i] == y[j]:  
                c[i,j] = c[i-1,j-1]+1 # take X[i] and Y[j] for LCS  
                b[i,j] = ADDXY  
            elif c[i-1,j] >= c[i,j-1]:  
                c[i,j] = c[i-1,j]  
                b[i,j] = SKIPY  
            else:  
                c[i,j] = c[i,j-1]  
                b[i,j] = SKIPX  
    return c, b
```

Example

- $X=\langle B,A,C,D,B\rangle$, $Y=\langle B,D,C,B\rangle$
 - $LCS=\langle B,C,B\rangle$ or $LCS=\langle B,D,B\rangle$
- BACDB
| | |
BDC-B
- BACD-B
| | |
B--DCB
- Two tied solutions can be generated with the preference of SKIPX or SKIPY consistently
- Insertion, deletion or mismatch have zero penalty

Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0					
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖				
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←			
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←		
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	1↖
D	0					
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	1↖
D	0	1↑	1←	1←	2↖	2←
C	0					
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	1↖
D	0	1↑	1←	1←	2↖	2←
C	0	1↑	1←	2↖	2←	2←
B	0					



Example (cont.)

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	1↖
D	0	1↑	1←	1←	2↖	2←
C	0	1↑	1←	2↖	2←	2←
B	0	1↖	1←	2↑	2←	3↖



Example (cont.)

- Locate the max value
|LCS|=3 and backtrack
- Follow the backtrack
arrows from (4,5) to (1,1)
to build the LCS
- ADDXY (\nwarrow) are the
matched elements

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1 \nwarrow	1 \leftarrow	1 \leftarrow	1 \leftarrow	1 \nwarrow
D	0	1 \uparrow	1 \leftarrow	1 \leftarrow	2 \nwarrow	2 \leftarrow
C	0	1 \uparrow	1 \leftarrow	2 \nwarrow	2 \leftarrow	2 \leftarrow
B	0	1 \nwarrow	1 \leftarrow	2 \uparrow	2 \leftarrow	3 \nwarrow



Example (cont.)

- Tie-breaker (\uparrow) will find the other LCS

		B	A	C	D	B
	0	0	0	0	0	0
B	0	1↖	1←	1←	1←	1↖
D	0	1↑	1↑	1↑	2↖	2←
C	0	1↑	1↑	2↖	2↑	2←
B	0	1↖	1↑	2↑	2↑	3↖



Sequence Alignment

- Local alignment
- Global alignment
 - End-to-end
- Alignment can be scored by the edit distance

- $$S(i,j) = \max \begin{cases} S(i-1,j-1) + \sigma(x_i, y_j) \\ S(i-1,j) + \sigma(x_i, -) \\ S(i,j-1) + \sigma(-, y_j) \end{cases}$$



Sequence Alignment

- Example global alignment

- ACGTAGGTATTTATGCTAGAT
| | | | | | | | | | | |
ACGT-----ATCTATGCT-GAT

- Example local alignment

- ...ACGTAGGTATTTATGCTAGAT...
| | | | | | | | | |
ACGTATCTATGCT-GAT



Implementation

```
def local_align(x[1..m], y[1..n], score):  
    A=Matrix(m,n)    # zero filled  
    best=0  
    optloc=(0,0)  
    for i in range(1,m+1):  
        for j in range(1,n+1):  
            A[i][j] = max(    # local alignment recurrence rule  
                A[i][j-1] + score.gap,    # insertion  
                A[i-1][j] + score.gap,    # deletion  
                A[i-1][j-1] +  
                    (score.match if x[i] == y[j] else score.mismatch),  
                0)    # local alignment  
            if A[i][j] >= best:    # track the matrix cell with the highest score  
                best = A[i][j]  
                optloc = (i,j)  
    return best, optloc
```



LCS versus Alignments

- A more general recurrence rule than LCS
- To calculate $\text{cell}[i,j]$,
take the max of $\{\text{cell}[i-1,j], \text{cell}[i-1,j-1], \text{cell}[i,j-1]\}$
- Algorithm is same except for the recurrence rule
- Considers insertions (\leftarrow), deletions (\uparrow), matches (\searrow) and mismatches (\swarrow) with reward and penalty scores



Local to Global Alignment

- Recurrence rule change
 - Remove "0" argument from the max operator
 - Negative values can take over due to insertions, deletions and mismatches
- Local Align score = $\max(\text{Global Align score}, 0)$
- Global alignment is end to end
 - i.e. corner to corner on the alignment matrix
- Local alignment can start (or end) at any (i, j)

