Introduction to Advanced Analysis Techniques

CS5012

Reading in the Algorithms textbook: Chapters 15 and 17

UNIVERSITY OF VIRGINIA
DATA SCIENCE INSTITUTE

Objectives of the Lecture

- Preview of ideas related to advanced analysis techniques.
- Encouragement to further study algorithms and other problem-solving paradigms
- Understanding the meaning, importance and applications of Advanced analysis Techniques, focusing on:
  - Dynamic Programming (DP)*
  - Greedy Algorithms**

* ***Focus***: introduction to DP [computing LCS]

** Previously covered

Dynamic Programming

- "Those who cannot remember the past are doomed to repeat it."
  -- George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense*

Dynamic Programming(DP): Introduction

- Dynamic programming is a way of solving complex problems by
  - dividing them into simpler sub problems, then
  - combining the solutions of sub problems to achieve an overall optimal solution

- The results of subproblems are stored to avoid working on the same subproblem again (*eliminating repetition*)
- This is called memoization *(not memorization!)*

Dynamic Programming: requirements

- The properties needed for a dynamic programming solution to be applicable are:
  - Overlapping subproblems. It must be possible to break the original problem down into subproblems, with some overlapping (some subproblems occur more than once)
  - Each of these subproblems will be solved only once, and the results saved and reused if necessary
  - Optimal substructure. It must be possible to calculate the optimal solution to a subproblem

- Solving the smaller problems leads to the final solution

DP: Overlapping subproblems

- A problem is said to have overlapping subproblems if
  - the problem can be broken down into subproblems which are reused several times
  - or a recursive algorithm for the problem solves the same subproblem over and over
- This issue of unnecessary repetition is handled well by dynamic programming

DP: Optimal Structure

- A given problem has an Optimal Substructure property if:

  - An optimal solution of the given problem can be obtained by using optimal solutions of its subproblems

DP: Comparison with Divide & Conquer

- Divide and Conquer algorithms partition the problem into **independent** subproblems
- In Dynamic Programming the subproblems are **not independent**
- Dynamic Programming algorithm solves every subproblem just once and then saves its answer in a table
- If a subproblem pops up again, it is NOT processed, instead the saved results are used
  - *Question: Is this the case for Divide and Conquer?*

DP: Comparison with Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment
- Often, greedy algorithms tend to be easier to code
- Greedy and Dynamic Programming, both, are methods for solving optimization problems
- Greedy algorithms are usually more *efficient* than DP solutions
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm

Examples of Algorithmic Paradigms

- Divide and Conquer
  - Divide into smaller sub-instances of the same problem, solve these recursively, and then put solutions together
  - Examples: Mergesort, Quicksort, Strassen's algorithm, FFT
- Greedy Algorithms
  - Make a choice that looks optimal at the moment —don't look ahead, never go back
  - Examples: Prim's algorithm, Kruskal's algorithm
- Dynamic Programming
  - Bottom up: find optimal solutions to subproblems ("turns recursion upside down")
  - Example: Floyd-Warshall algorithm for the all pairs shortest path problem

DP: *Some areas of application*

- Matrix multiplication
- Bioinformatics
- DNA analysis
- Control theory
- Information theory
- Operations research
- Computer science: AI, theoretical analyses, graphics and many more

DP: *Example Algorithms*

- Computing a binomial coefficient
- Longest common subsequence
- Warshall's algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - Traveling salesman
  - Knapsack

DP: *Example Algorithms*

- Viterbi for hidden Markov models
- Unix `diff` for comparing two files
- Smith-Waterman for sequence alignment
- Bellman-Ford for shortest path routing in networks
- Cocke-Kasami-Younger for parsing context free grammars

DP: Improving the efficiency

- Often when using a more naive method, many of the subproblems are generated and *solved many times*
- The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations

- By eliminating unnecessary repetitions, Dynamic programming can bring down the order of time complexity to $O(n^2)$ or $O(n^3)$ instead of being exponential
- Sometimes even better than $O(n^2)$

DP: Four general steps

1. Characterize the structure of an optimal solution
2. Define the optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion by solving the subproblems (some overlapping)
4. Construct an optimal solution from computed information

Three key elements of DP algorithms

- Substructure: Decompose the problem into smaller sub-problems. Express the solution of the original problem in terms of solutions for smaller problems

- Table-structure: Store the answers to the sub-problem in a **table**, because sub-problem solutions may be used many times

- Bottom-up computation: Combine solutions on smaller sub-problems to solve larger sub-problems, and eventually arrive at a solution to the complete problem

Example1:
Longest Common Subsequence (LCS)

Definition is as follows:

- Given **two strings**:
  - string S of length ***n***, and
  - string T of length ***m***

(Length of individual strings do not matter. "n" and "m" could be the same, or they could be different)

- The goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings

Try It! ~ A simple example

- Let there be string1: S = ABAZDC
- Let there be string 2: T = BACBAD

- What is the longest common subsequence? (and how long is it?)

Example1: LCS – A simple example

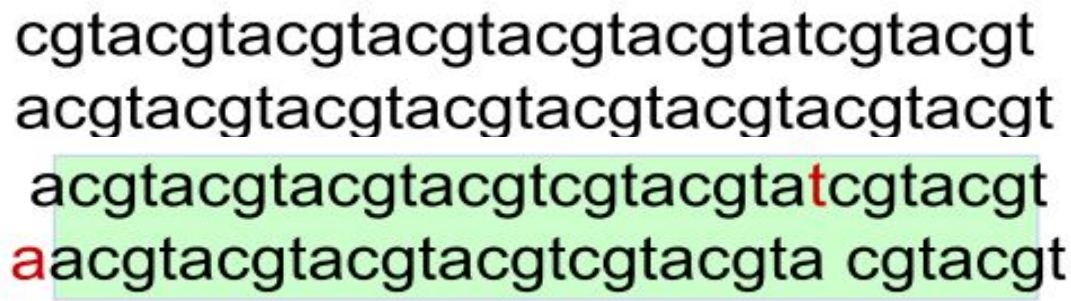
- Let there be string1: S = ABAZDC
- Let there be string 2: T = BACBAD

- In this case, the LCS has length 4 and is the string **ABAD**.

  Another way to look at it is we are finding a one-to-one matching between *some* of the letters in S and *some* of the letters in T

Example1: Motivation

- Approximate string matching [Levenshtein, 1966]
  - Search for “occurance”, get results for “occurrence”
- Computational biology [Needleman-Wunsch, 1970’s]
  - Simple measure of genome similarity



  - n-length(LCS(x,y)) is called the “edit distance”

Example1: Further Applications of LCS

- LCS have many applications. Here are some of the areas:
  - Molecular biology, DNA sequences
  - File comparison
  - Screen redisplay

Example1: Use of LCS in Molecular biology

- DNA sequences (genes) can be represented as sequences of four letters **ACGT**, corresponding to the four sub-molecules forming DNA
- Comparison of two sequences (human beings, animals or plants) by finding LCSs we can find
  - patterns of diseases
  - closeness of relationship among individuals
  - cultural heritage
  - relation of a crime scene to an individual
  - etc …

Example1: Use of LCS in File Comparison

- The Unix program "`diff`" is used to compare two different versions of the same file, to determine what changes have been made to the file
- It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed

Example1: Use of LCS in Screen Redisplay

- Many text editors like "**emacs**" display part of a file on the screen, updating the screen image as the file is changed
- For slow dial-in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. It is possible to view the computation of the minimum length sequence of characters needed to update the terminal as being a sort of common subsequence problem

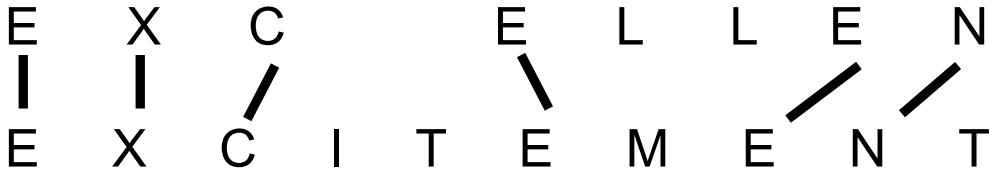
Example1: LCS – Another example

- A subsequence of a sequence/string S is obtained by deleting zero or more symbols from S
- For example, the following are some subsequences of "president": pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in S, but they are not required to be consecutive

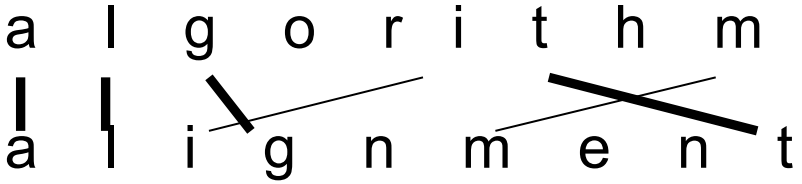
- The longest common subsequence (LCS) problem is to find a maximum length common subsequence between two sequences

Example1: If the lines cross, don't add the symbol

S 1: EXCELLENT, S 2: EXCITEMENT, LCS = EXCEEN



S 1: ALGORITHM, S 2: ALIGNMENT, LCS = ALGT



Note the crossed lines. The 2nd symbol that cause the lines to be crossed is **not** added

In the example above, "**i**" and "**m**" are <u>not</u> added

Example1: LCS – Brute Force Solution

Solution: For every subsequence of ***x***, check if it is a subsequence of ***y***

Analysis :

- There are 2^m subsequences of ***x***
- Each check takes O(n) time, since we scan ***y*** for first element, and then scan for second element, etc.
- The worst case running time is $O(n2^m)$ = *Exponential!*
- Example – For S 1: algorithm, S 2: alignment = $9*2^9$ = 4608 comparisons!]

Example1: Formally computing LCS

Let A=$a_1a_2\ldots a_m$ and $B=b_1b_2\ldots b_n$.

- *len*(*i, j*): the length of an LCS between
  $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$
- With proper initializations, *len*(*i, j*) can be computed as follows.

$$len(i,j)=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ len(i-1,j-1)+1 & \text{if } i,j>0 \text{ and } a_i=b_j,\\ \max(len(i,j-1),len(i-1,j)) & \text{if } i,j>0 \text{ and } a_i\neq b_j.\end{cases}$$

Example1: Sample code Computing LCS

```
procedure LCS-Length(A, B)
1.  for i ← 0 to m do len(i,0) = 0
2.  for j ← 1 to n do len(0,j) = 0
3.  for i ← 1 to m do
4.      for j ← 1 to n do
5.          if a_i = b_j then [ len(i, j) = len(i − 1, j − 1) + 1
                              [ prev(i, j) =" ↖ "
6.      else if len(i − 1, j) ≥ len(i, j − 1)
7.          then [ len(i, j) = len(i − 1, j)
                 [ prev(i, j) =" ↑ "
8.          else [ len(i, j) = len(i, j − 1)
                 [ prev(i, j) =" ← "
9.  return len and prev
```

If letters are the same; **add 1** to upper left diagonal and draw arrow

If letters are **not** the same; if box above len greater than box to the left, point UP

To break a tie (up and left have same len) then point UP!

If letters are **not** the same; if box to the left len greater than box to the top, point to the LEFT

Example1:
Computing LCS using the table

i		j	0	1	2	3	4	5	6	7	8	9	10
				p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0			0	0	0	0	0	0	0	0	0	0	0
1	*p*		0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	*r*		0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	*e*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	*s*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	*i*		0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	*d*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	*e*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	*n*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	*t*		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Running time and memory: ***O(mn)*** *and* ***O(mn)***

Example1:
The Backtracking Algorithm

- Apply the following algorithm to produce the output. Begin at the bottom right of the table, working backwards

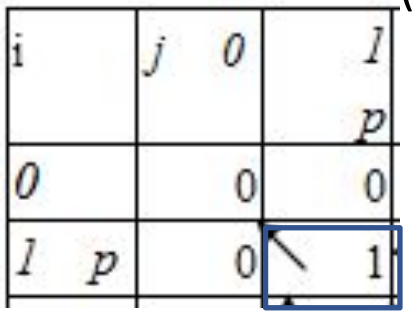
```
procedure Output-LCS(A, prev, i, j)
1   if i = 0 or j = 0 then return
2   if prev(i, j)="↖" then [Output − LCS(A, prev, i − 1, j − 1)
                            print a_i
3   else if prev(i, j)="↑" then Output-LCS(A, prev, i-1, j)
4   else Output-LCS(A, prev, i, j-1)
```

On Diagonals Print the Letter!

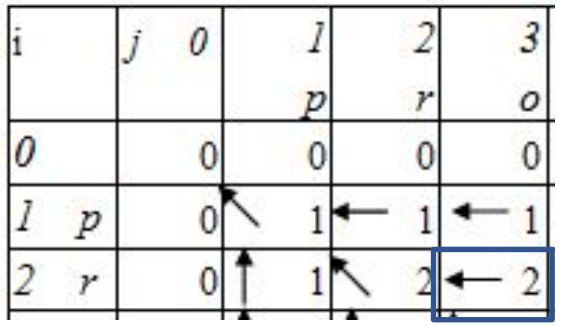
- Final output is just imply reversed:
  - Utilizing this algorithm you'll get: `nedirp`
  - Reversing it, the final output is: `priden`

Example1: The Backtracking Algorithm

- If the letters match (“p”&“p”), add one to the diagonal (0,0) and enter that value (**1**) in the box (1,1). Add a diagonal arrow: [*See bottom right corner*]



- If the letters do not match (“r”&“o”), pick the largest number of the diagonals (2,2) or (1,3) and draw an left arrow to the larger number. Will enter the
(If diagonals are *equal* pick the top)
- [*See bottom right corner*]

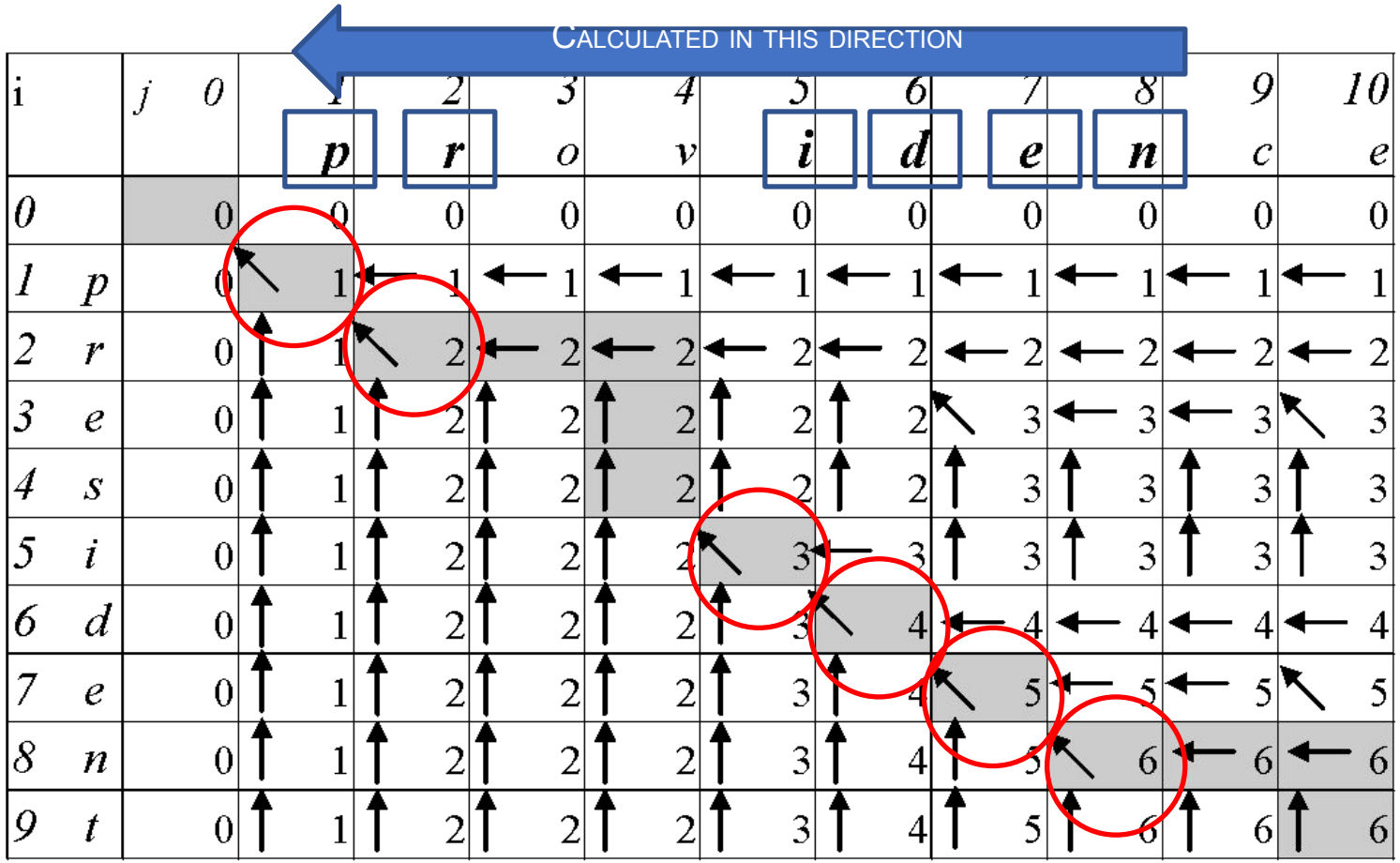


Computing LCS using the table

i		j	0	1	2	3	4	5	6	7	8	9	10
				p	*r*	*o*	*v*	*i*	*d*	*e*	*n*	*c*	*e*
0			0	0	0	0	0	0	0	0	0	0	0
1	*p*		0	1	1	1	1	1	1	1	1	1	1
2	*r*		0	1	2	2	2	2	2	2	2	2	2
3	*e*		0	1	2	2	2	2	2	3	3	3	3
4	*s*		0	1	2	2	2	2	2	3	3	3	3
5	*i*		0	1	2	2	2	3	3	3	3	3	3
6	*d*		0	1	2	2	2	3	4	4	4	4	4
7	*e*		0	1	2	2	2	3	4	5	5	5	5
8	*n*		0	1	2	2	2	3	4	5	6	6	6
9	*t*		0	1	2	2	2	3	4	5	6	6	6

Output: *priden*

Computing LCS using the table



i		j	0	1	2	3	4	5	6	7	8	9	10
				p	**r**	o	v	**i**	**d**	**e**	**n**	c	e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	1	1	1	1	1	1	1	1	1	1
2	r		0	1	2	2	2	2	2	2	2	2	2
3	e		0	1	2	2	2	2	2	3	3	3	3
4	s		0	1	2	2	2	2	2	3	3	3	3
5	i		0	1	2	2	2	3	3	3	3	3	3
6	d		0	1	2	2	2	3	4	4	4	4	4
7	e		0	1	2	2	2	3	4	5	5	5	5
8	n		0	1	2	2	2	3	4	5	6	6	6
9	t		0	1	2	2	2	3	4	5	6	6	6

Output: *priden*

Example2: Finding the nth Fibonacci number

- In this example we will discuss how the time complexity of finding the nth Fibonacci number can be dramatically improved by using dynamic programming

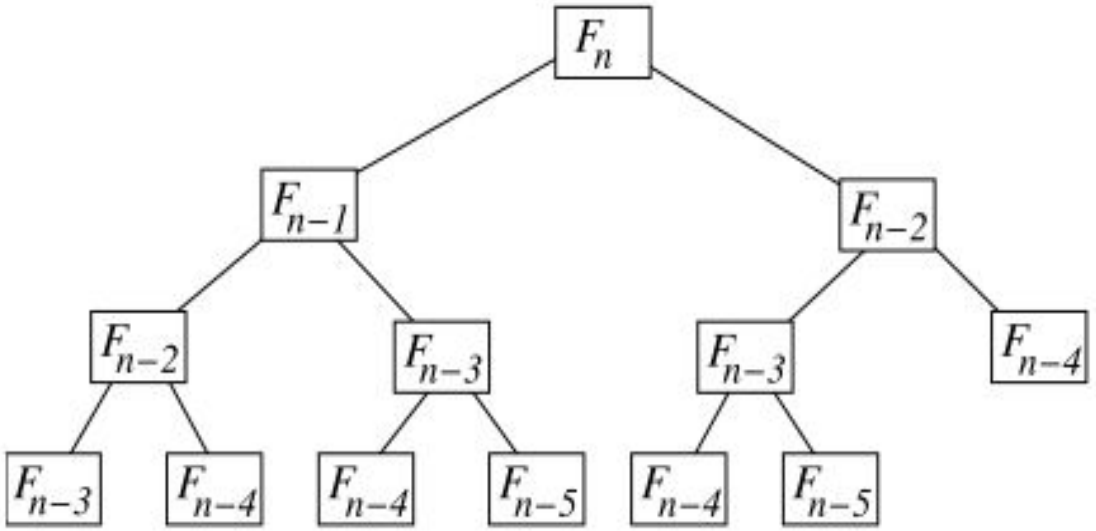
Example2: Finding the nth Fibonacci number

- Fibonacci Numbers:
  Basic rules for calculatio

$F_0 = 0,$

$F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$ (for *n*



- A recursive algorithm

Algorithm REC-FIB(n)

1. **if** $n = 0$ **then**
2. **return** 0
3. **else if** $n = 1$ **then**
4. **return** 1
5. **else**
6. **return** REC-FIB($n-1$)+ REC-FIB($n-2$)

Very slow! Exponentially
many repeated computations

Example2: Finding the 5th Fibonacci number

- Let us say that we want to find the **5th** Fibonacci number
- RULES: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

Number:	0	1	2	3	4	5
Fib_n:	0	1	1	2	3	5

Example2: Finding the 5th Fibonacci number

- Let us look, once again, at the properties needed for a dynamic programming solution to be applicable are:

1. Overlapping subproblems
2. Optimal substructure

- Re: 1 – to find Fib(5), we must find Fib(4), to find Fib(4) we must find Fib(3) and so on…
- Re: 2 – if our solutions to subproblems (say Fin(3) and Fib(4)) are optimal the final solution will also be optimal

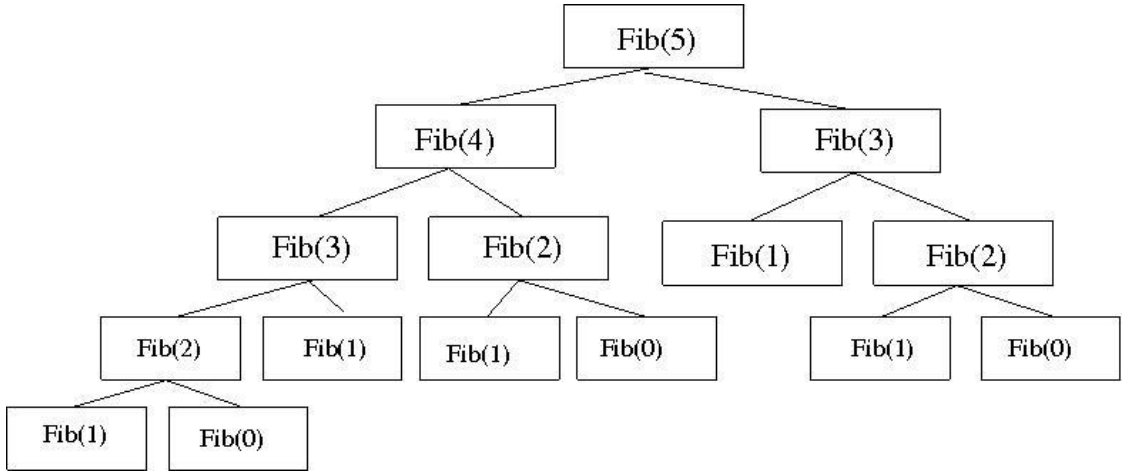
Tracing Recursive Rec-Fib(5)

- For Rec-Fib(5), we call **Rec-Fib(4)** and **Rec-Fib(3)**
  - For **Rec-Fib(4)**, we call Rec-Fib(3) and Rec-Fib(2)
    - For Rec-Fib(3), we call Rec-Fib(2) and Rec-Fib(1)
      - For Rec-Fib(2), we call Rec-Fib(1) and Rec-Fib(0). Base cases!
      - Rec-Fib(1). *Base case!*
    - For Rec-Fib(2), we call Rec-Fib(1) and Rec-Fib(0). Base cases!
  - For **Rec-Fib(3)**, we call Rec-Fib(2) and Rec-Fib(1)
    - For Rec-Fib(2), we call Rec-Fib(1) and Rec-Fib(0). Base cases!
    - Rec-Fib(1). *Base case!*

Note that subproblems (like **Rec-Fib(2)**) repeat, and solved again and again
Don't remember that the subproblem has been solved before
 Better to store partial solutions instead of recalculating values repeatedly

Example2: Finding the 5th Fibonacci number

- Below, we can observe the overlapping to compute Fib(5): Calculating Fib(3) twice, Fib(2) thrice and Fib(1)five times
- This effort can be saved by computing only once and re-using the results. This technique is called memoization and is used in Dynamic Programming



Example2: Finding the 5th Fibonacci number

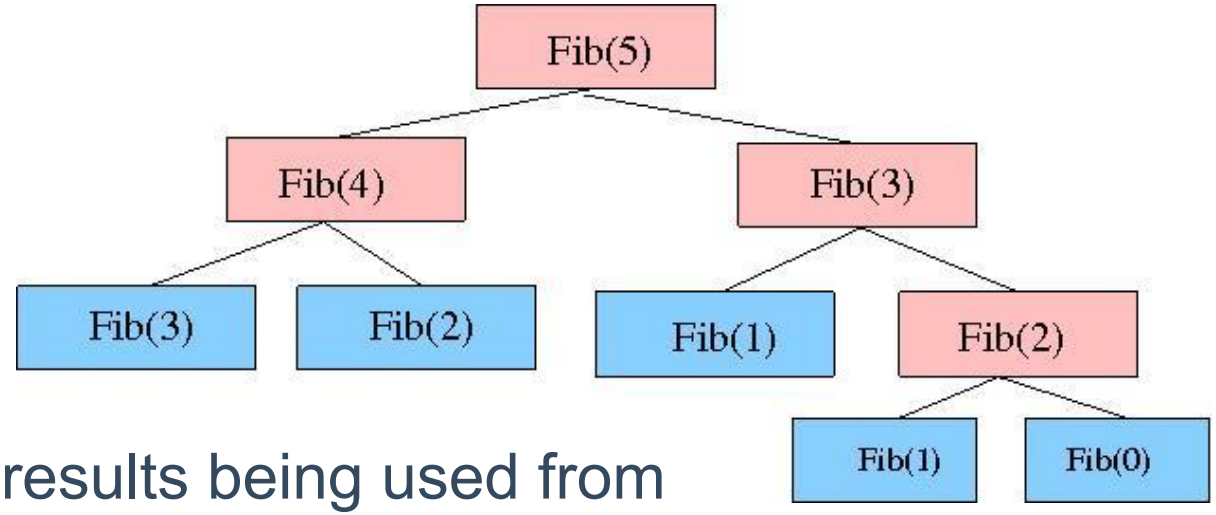
```
Fib(n) {
  if (n == 0)     return M[0];
  if (n == 1)  return M[1];
  if (Fib(n-2) is not already calculated)
       call Fib(n-2); // & store result
  if(Fib(n-1) is already calculated)
       call Fib(n-1); // & store result
//Store the nth Fibonacci# in memory & use prev. results
  M[n] = M[n-1] + M[n-2]
  Return M[n];
}
```

Example2: Finding the 5th Fibonacci number

- In the previous code of dynamic programming, instead of calling Fib(n−1) and Fib(n−2) we are first checking if they are already present in memory or not. And each computation stores the result in memory for future references
- If a computation has already been done before, the program fetches the stored results and uses them

Example2: Finding the 5th Fibonacci number

- The tree for this new algorithm will look something like this:



Here **blue** boxes represent results being used from memory and **pink** boxes represent results being computed

DP: Additional Solution to calculate Fibonacci Numbers

- Dynamic Programming Approach [Solution #1]

Algorithm DYN-FIB(n)

1. $F[0] = 0$
2. $F[1] = 1$
3. **for** $i \leftarrow 2$ **to** n **do**
4. $\quad F[i] \leftarrow F[i-1] + F[i-2]$
5. **return** $F[n]$

- Build "from the bottom up"
- Running time: O(n)
- Very fast in practice – just need an array (of linear size) to store the F(i) values

DP: Additional Solution to calculate Fibonacci Numbers

- Dynamic Programming Approach [Solution #2]

```
long fib(int n) {
  if ( n < 2 ) return 1;
    long answer;
    long prevFib=1, prev2Fib=1; // fib(0) & fib(1)
    for (int k = 2; k <= n; ++k) {
            answer = prevFib + prev2Fib;
            prev2Fib = prevFib;
            prevFib = answer;
  }
  return answer;
}
```

- Also built "from the bottom up." Running time: O(n)
- Very fast in practice – Storing *just* the previous two Fibonacci numbers (*saving much more space!*)

Example2: Final Notes

- The bigger problem is broken down into a series of smaller sub-problems
- To prevent calculating again and again, the results for the smaller subproblems are saved and reused when needed
- Results of the subproblems lead to the optimal solution for the final problem

Additional Slides

- Supplemental material

Example3: Multiplication of Matrices

- Let us take a simple example of multiplication of 3 matrices: M1, M2 and M3
- There have 2 options:
  - First multiply M1 and M2, and then multiply the results with M3: (M1.M2) (M3)
  - First multiply M2 and M3, and then multiply the results with M1: (M1) (M2.M3)

Example3: Notes on Matrix Multiplications

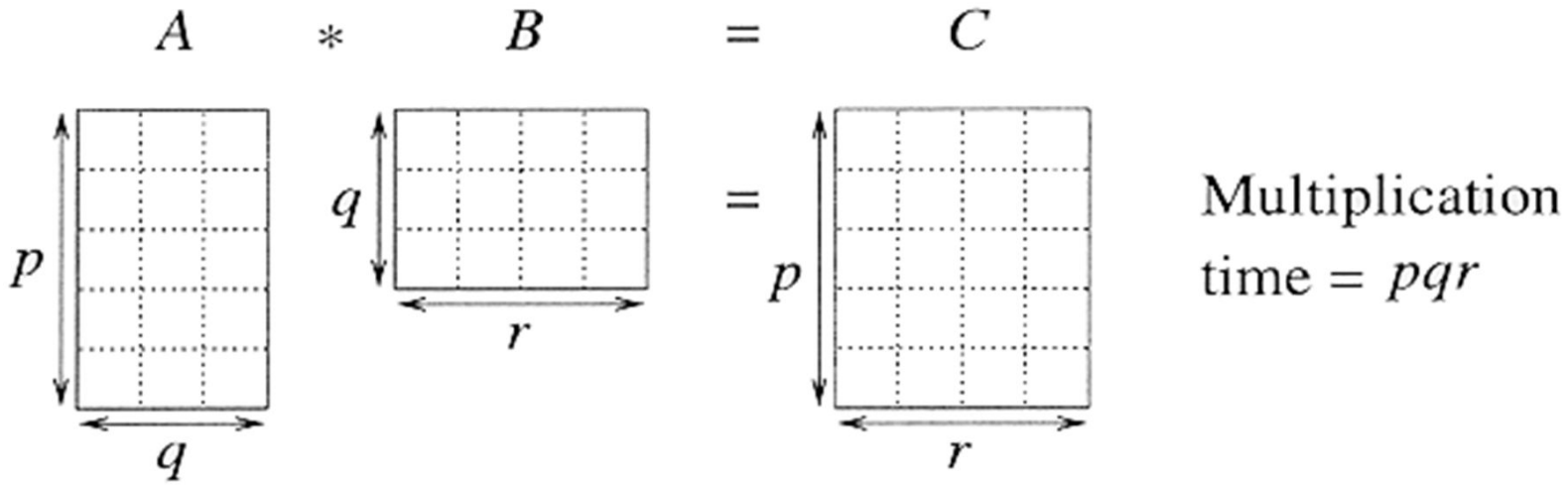
- Two matrices can only be multiplied if they are compatible: the of columns of the first matrix must be equal to the number of rows of the second matrix
  If M1 = pxq then M2 = ***must*** be qxr

- To multiply M1 and M2, the total number of multiplications involved is ***pqr***

- The dimensions of the resulting matrix (M1.M2) are **p x r**

Product ‘C’ resulting from A . B

- Note that there are ***pr*** total entries in C and each takes O(q) time to compute, thus the total time to multiply 2 matrices is ***pqr*** (*also number of multiplications/operations*)



Matrix-Chain Multiplication

- Matrix multiplication is associative, so all parenthesizations yield the same product
- For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then the following multiplications are equivalent:

  (A1 (A2 (A3 A4)))
  (A1 ((A2 A3) A4))
  ((A1 A2) (A3 A4))
  ((A1 (A2 A3)) A4)
  (((A1 A2)A3) A4)

- Above 5 methods will bring the same final result, but the number of calculations (multiplications) will be different

Steps involved for (M1.M2) (M3)

Let us assume:
M1 = 10 x 100, M2 = 100 x 5 and M3 = 5 x 50

- M1= 10 x100 and M2 = 100 x 5

  pqr = 10x100x5 = **5000** *(number of multiplications)*

  The product matrix of M1.M2 is p x r = 10 x 5

- Now we multiply resulting 10 x 5 matrix with M3 (5x50)

  pqr = 10 x 5 x 50 = **2500** *(number of multiplications)*

- Total number of calculations: 5000 + 2500 = **7500**

Steps involved for (M1) (M2.M3)

Reminder:
M1 = 10 x 100, M2 = 100 x 5 and M3 = 5 x 50

- M2= 100 x 5 and M3 = 5 x 50

  pqr = 100x5x50 = **25000** *(multiplications)*

  The product matrix of M2.M3 is p x r = 100 x 50

- Now we multiply resulting 100 x 50 matrix with M1 (10x100)

  pqr = 10 x 100 x 50 = **50000** *(multiplications)*

- Total number of calculations: 25000 + 50000 = **75000**

Example3: Final Notes

- Multiplying as (M1.M2) (M3) needs only 7500 calculations whereas multiplying as (M1) (M2.M3) needs 75000 calculations
- Our example makes it clear that different multiplication orders do **not** cost the same
- Time complexity of the first method is only a fraction of the second method
- The difference becomes even more obvious when a large number of matrices are to be multiplied

Example3: Matrix Multiplication

- From the previous calculations, using a very simple example, it becomes very clear that time complexity shows dramatic difference even for multiplying only three matrices
- Obviously, it is unwise to try any large number of multiplication using a naïve algorithm
- Dynamic programming emphasizes the concept of optimal structure. In this case, an optimal sequence of matrix multiplications resulting in smallest number of operations
  - Structure relates to the parenthesis (determining sequence of multiplications)

Example3: Matrix Multiplication

- Let's find the best way to multiply ***n*** number of matrices: $A_1, A_2, A_3, \ldots, A_n$
- Let us remember that multiplying p x q matrix A and q x r matrix B takes ***p q r*** multiplications and the resulting matrix is p x r
- In general, let A_i be p_{i-1} x p_i matrix
- P_i is the number of columns of matrix ***i***. The rows of matrix i are equal to the columns of matrix p_{i-1}
- $A_1, A_2, A_3, \ldots$, can can be represented by $p_0; p_1; p_2; p_3, \ldots, p_n$
- Let m(i, j) denote minimal number of multiplications needed to compute $A_i \; A_{i+1} \; A_j$
    - We want to compute **m(1; n)**

Example3: Matrix Multiplication

Divide and conquer solution/ recursive algorithm

- Let's assume the position of last product is ***k***
- Then we have to compute recursively the best way to multiply chain from **i to k**, and from **k+1 to j** and add the cost of the final product
- This means that:
  - $m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} \cdot p_k \cdot p_j$
- If we don't know ***k***, then we have to try all possible values of k and pick the best solution
- To know all possible values we recursive formulation of m(i,j)

$$m(i,j) = \begin{cases} 0 & \text{If } i = j \\ \min_{i \le k < j} \{m(i,k) + m(k+1,j) + p_{i-1} \cdot p_k \cdot p_j\} & \text{If } i < j \end{cases}$$

Example3: Coded form of the recursive formulation

Matrix-chain(i, j)
  IF i = j THEN return 0
  m = ∞
  FOR k = i TO j-1 DO
    q = Matrix-chain(i, k) + Matrix-chain(k + 1, j) + $p_{i-1} \cdot p_k \cdot p_j$
    IF q < m THEN m = q
  OD
  Return m
END Matrix-chain

Return Matrix-chain(1, n)

Running time of previous program

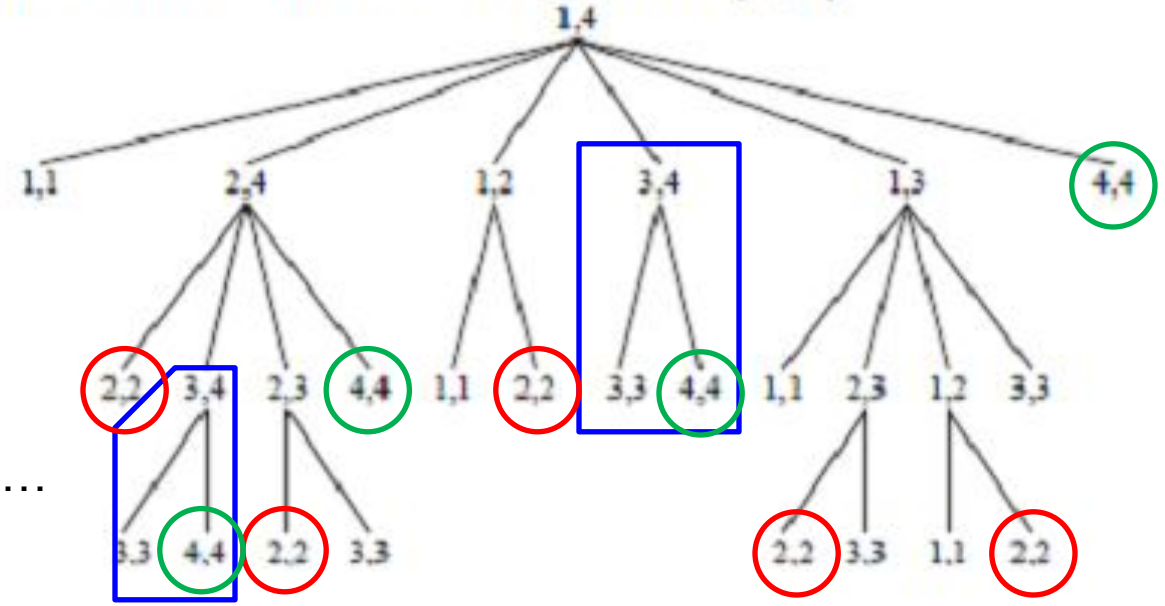
$$
\begin{aligned}
T(n) &= \sum_{k=1}^{n-1} (T(k) + T(n-k) + O(1)) \\
&= 2 \cdot \sum_{k=1}^{n-1} T(k) + O(n) \\
&\geq 2 \cdot T(n-1) \\
&\geq 2 \cdot 2 \cdot T(n-2) \\
&\geq 2 \cdot 2 \cdot 2 \ldots \\
&= 2^n
\end{aligned}
$$

Time complexity is exponential

Processing gets very slow as ***n*** increases

Duplications of calculations

- Problem is that we compute the same result over and over again.
  - Example: Recursion tree for MATRIX-CHAIN(1, 4)



Imagine if this tree was bigger, the number of repetitions would be greater

We compute the matrix chain (3,4) twice, …

This waste can be *avoided*

Example3: Matrix Multiplication

- To avoid recalculating that has already been done before, we store the results. The process is called memoization
- Initially all entries are set to ∞

```
FOR  i = 1 to n DO
     FOR j = i to n DO
       T [ i ][ j ] = ∞
     OD
OD
```

- The code for MATRIX-CHAIN (i,j) stays the same, except that pre-existing table is checked before to see whether T[i] [j] is already computed. If the result is available in the table, it is returned otherwise calculations are done and saved in the table

Revised code to prevent duplicate processing

```
MATRIX-CHAIN(i, j)
    IF T[i][j] < ∞ THEN return T[i][j]
    IF i = j THEN T[i][j] = 0, return 0
    m = ∞
    FOR k = i to j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    T[i][j] = m
    return m
END MATRIX-CHAIN

return MATRIX-CHAIN(1, n)
```

The look up table prevents duplicate calculations

Example3: Matrix Multiplication Running Time (using DP)

- Different calls to MATRIX-CHAIN (i,j)
- The first time a call is made it takes O(n) time, not counting recursive calls
- When a call has been made once, it cost O(1) time to make it again

---

(Additional information is available as supplemental material at the end of these slides)

Matrix-Chain Multiplication

matrix	dimension
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 100 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= (7125)$$

DP Solution

- The optimal cost can be described be as follows:
  - $i = j \Rightarrow$ the sequence contains only 1 matrix, so $m[i, j] = 0$
  - $i < j \Rightarrow$ This can be split by considering each k, $i \leq k < j$,
    as $A_{i \ldots k}$ (p_{i-1} x p_k) times $A_{k+1 \ldots j}$ (p_k x p_j)

- This suggests the following recursive rule for computing m[i, j]:

$m[i, i] = 0$

$m[i, j] = \min_{i \leq k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$ for $i < j$

Computing *m[i, j]*

- For a specific k,

  $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$

  $=$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Computing *m[i, j]*

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$
$$= A_{i \ldots k}(A_{k+1} \ldots A_j)\ (m[i, k] \text{ mults})$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Computing *m[i, j]*

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$
$$= A_{i \ldots k}(A_{k+1} \ldots A_j) \; (m[i, k] \text{ mults})$$
$$= A_{i \ldots k} A_{k+1 \ldots j} \quad (m[k+1, j] \text{ mults})$$

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$

Computing *m[i, j]*

- For a specific k,

$$
\begin{aligned}
&(A_i \ldots A_k)(A_{k+1} \ldots A_j) \\
&= A_{i \ldots k}(A_{k+1} \ldots A_j) \; (m[i, k] \text{ mults}) \\
&= A_{i \ldots k} A_{k+1 \ldots j} \quad (m[k+1, j] \text{ mults}) \\
&= A_{i \ldots j} \quad (p_{i-1} p_k p_j \text{ mults})
\end{aligned}
$$

$$
m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)
$$

Computing *m[i, j]*

- For a specific k,

$$(A_i \ldots A_k)(A_{k+1} \ldots A_j)$$
$$= A_{i \ldots k}(A_{k+1} \ldots A_j) \ (m[i, k] \text{ mults})$$
$$= A_{i \ldots k} A_{k+1 \ldots j} \quad (m[k+1, j] \text{ mults})$$
$$= A_{i \ldots j} \ (p_{i-1} p_k p_j \text{ mults})$$

- For solution, evaluate for all k and take the *minimum*

$$m[i, j] = \min_{i \le k < j} (m[i, k] + m[k+1, j] + p_{i-1} p_k p_j)$$