MA 252: Data Structures and Algorithms

Lecture 32

http://www.iitg.ernet.in/psm/indexing_ma252/y12/index.html

Partha Sarathi Mandal

Dept. of Mathematics, IIT Guwahati

The All-Pairs Shortest Paths Problem

- Given a weighted digraph $G = (V, E)$ with weight function $w: E \rightarrow R$, (R, is the set of real numbers) determine the length of the **shortest path** i.e., distance between **all pairs of vertices** in G. Here we assume that there are no cycles with zero or negative cost.

SSSP

- Dijkstra's SSSP algorithm requires *all* edge weights to be nonnegative
  - even more restrictive than outlawing negative weight cycles
  - Runtime O(E log V) if priority queue is implemented with a binary heap.
  - Runtime O(V log V + E) priority queue is implemented with a Fibonacci heap.
- Bellman-Ford SSSP algorithm can handle negative edge weights
  - even "handles" negative weight cycles by reporting they exist
  - Runtime O(V E)

All pairs shortest paths

- Simple approach
  - Call Dijkstra's $|V|$ times
  - $O(|V|\,|E| \log |V|)$ / $O(|V|^2 \log |V| + |V|\,|E|)$
  - Call Bellman-Ford $|V|$ times
  - $O(|V|^2\,|E|)$
- A dynamic programming solution. Only assumes no negative weight cycles.
  - First version is $\Theta(|V|^4)$
  - Repeated squaring reduces to $\Theta(|V|^3 \log |V|)$
- Floyd-Warshall – $\Theta(|V|^3)$
- Johnson's algorithm – $O(|V|^2 \log |V| + |V|\,|E|)$

Dynamic programming

Computing Fibonacci Numbers

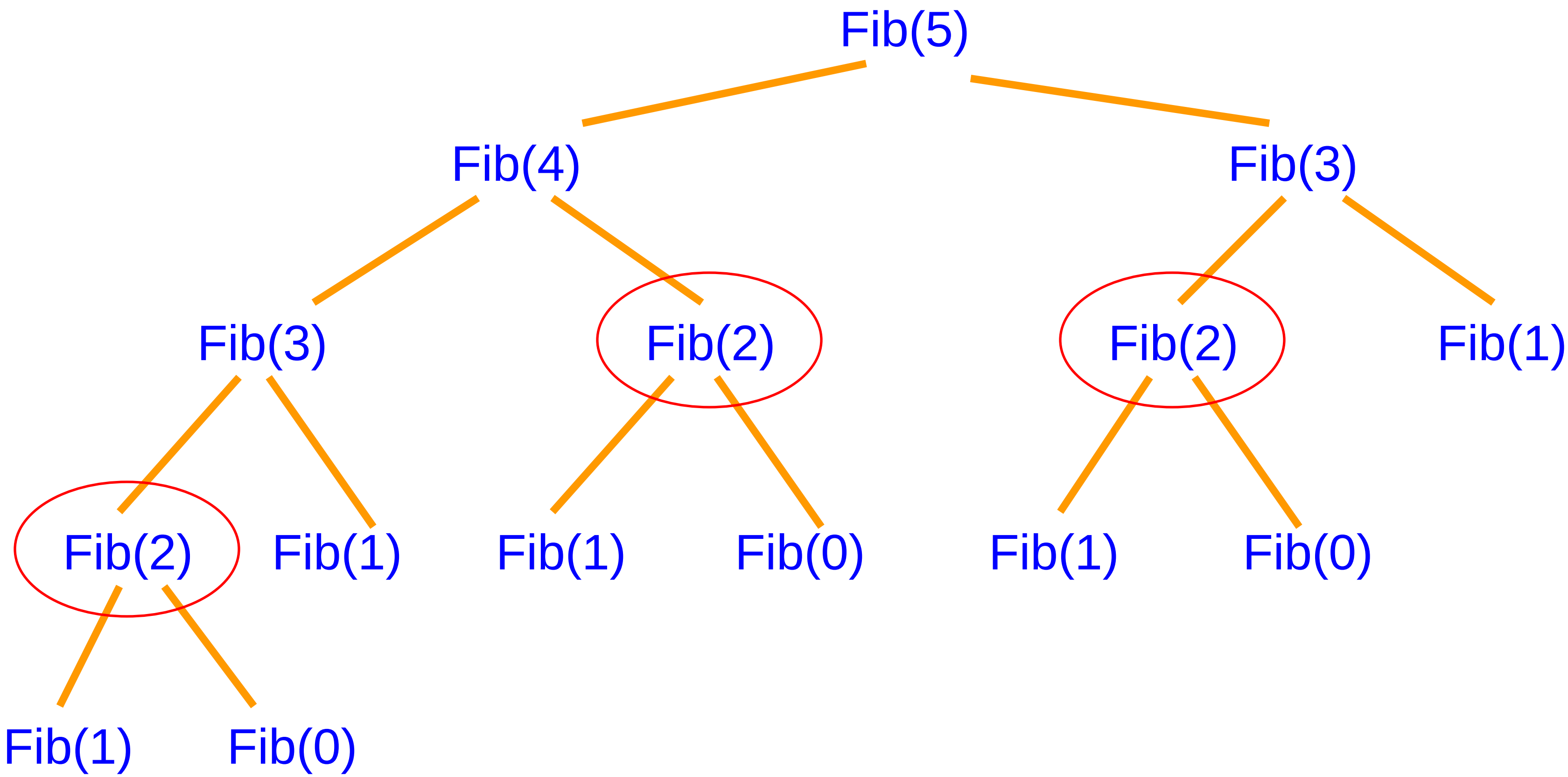
- **Fibonacci numbers:**
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- Sequence is 0, 1, 1, 2, 3, 5, 8, 13, …

Computing Fibonacci Numbers

- Obvious recursive algorithm:

- Fib(n):
  - if $n = 0$ or 1 then return n
  - else return (Fib(n-1) + Fib(n-2))

Recursion Tree for Fib(5)



How Many Recursive Calls?

- If all leaves had the same depth, then there would be about 2^n recursive calls.
- But this is over-counting.
- However with more careful counting it can be shown that it is $\Omega((1.6)^n)$
- Exponential!

Save Work

- Wasteful approach - repeat work unnecessarily
  - Fib(2) is computed **three** times
- Instead, compute Fib(2) once, store result in a table, and access it when needed

More Efficient Recursive Algo

- F[0] := 0; F[1] := 1; F[n] := Fib(n);
- Fib(n):
  - if n = 0 or 1 then return F[n]
  - if F[n-1] = NIL then F[n-1] := Fib(n-1)
  - if F[n-2] = NIL then F[n-2] := Fib(n-2)
  - return (F[n-1] + F[n-2])

Called memorization

- computes each F[i] only once

Example of Memoized Fib

	F
0	0
1	1
2	NIL 1
3	NIL 2
4	NIL 3
5	NIL 5

Fib(5) — fills in F[4] with 3, returns 3+2 = 5

Fib(4) — fills in F[3] with 2, returns 2+1 = 3

Fib(3) — fills in F[2] with 1, returns 1+1 = 2

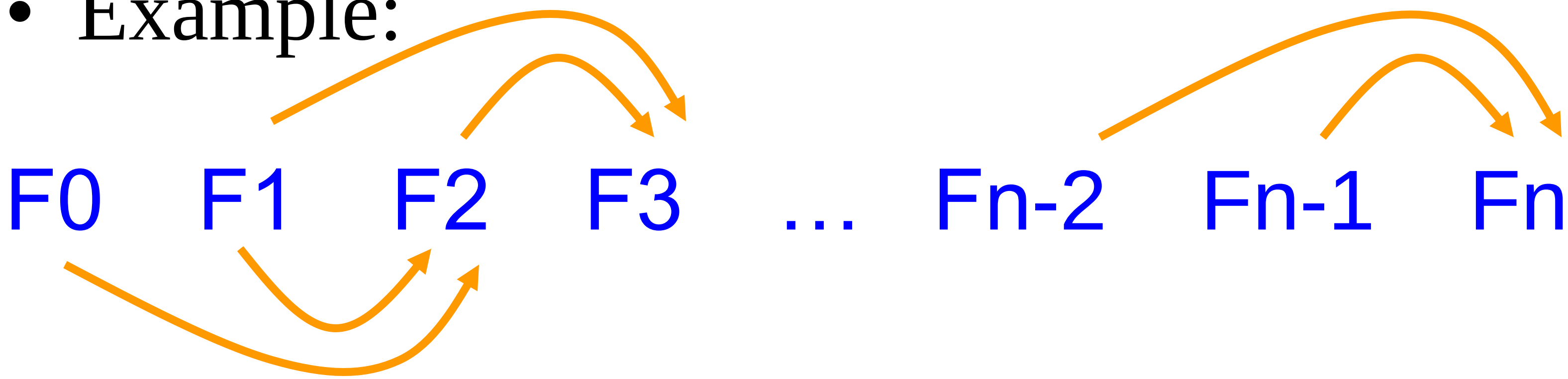
Fib(2) — returns 0+1 = 1

Get Rid of the Recursion

- Recursion adds overhead
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- Avoid the recursion overhead by filling in the table entries **bottom up**, *instead* of **top down**.

Subproblem Dependencies

- Figure out which subproblems rely on which other subproblems
- Example:



Order for Computing Subproblems

- Then figure out an order for computing the subproblems that respects the dependencies:
  - when you are solving a subproblem, you have already solved all the subproblems on which it depends
- Example: Just solve them in the order

$F_0, F_1, F_2, F_3, \ldots$

called Dynamic Programming

DP Solution for Fibonacci

- Fib(n):
  - F[0] := 0; F[1] := 1;
  - for i := 2 to n do
    - F[i] := F[i-1] + F[i-2]
  - return F[n]
- Can perform application-specific optimizations
  - e.g., save space by only keeping last two numbers computed

Time reduced from exponential to linear

Dynamic Programming (DP) Paradigm

- DP is typically applied to Optimization problems.
- DP can be applied when a problem exhibits:
- Optimal substructure:
  – Is an optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems:
  – If recursive algorithm revisits the same problem over and over again.

Dynamic Programming (DP) Paradigm

- DP can be applied when the solution of a problem includes solutions to subproblems
- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- In the end we'll get the solution of the whole problem

Dynamic programming

- One of the most important algorithm tools!
- Very common interview question

- Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems AND
- the sub-problems are overlapping

Identifying a dynamic programming problem

- The solution can be defined with respect to solutions to subproblems

- The subproblems created are overlapping, that is we see the same subproblems repeated

Two main ideas for dynamic programming

- Identify a solution to the problem with respect to **smaller** subproblems
  - $F(n) = F(n-1) + F(n-2)$
- Bottom up: start with solutions to the smallest problems and build solutions to the larger problems

```
Fib(n):
    F[0] := 0; F[1] := 1;
    for i := 2 to n do
        F[i] := F[i-1] + F[i-2]
    return F[n]
```

use an array to store solutions to subproblems

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ABA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ABA

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

ACA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ACA

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

$$X = A\ B\ A\ C\ D\ A\ B\ A\ B$$

DCA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

~~DCA~~

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

AADAA?

Longest common subsequence (LCS)

- For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

AADAA

LCS problem

- Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y
- Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, What is the **longest** common subsequence?

$$X = A\ B\ C\ B\ D\ A\ B$$

$$Y = B\ D\ C\ A\ B\ A$$

LCS problem

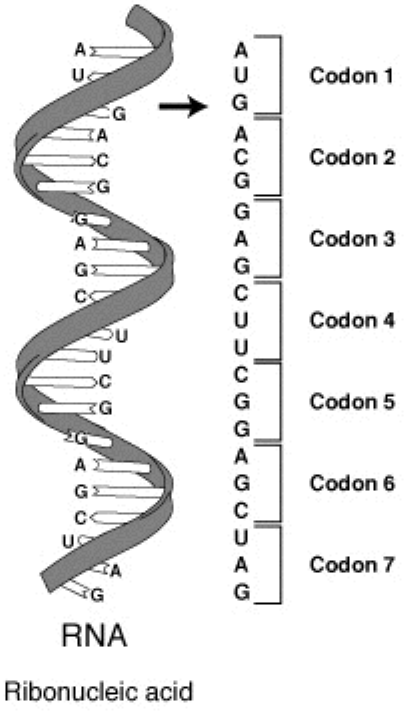
- Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y
- Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, What is the **longest** common subsequence?

X = A B C B D A B

Y = B D C A B A

The sequences{B, D, A, B} of length 4 are the LCS of *X* and *Y*, since there is no common subsequence of length 5 or greater.

LCS problem



Application:

comparison of two DNA strings

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- **Brute-force algorithm:** For every subsequence of x, check if it's a subsequence of y
  - *How many subsequences of x are there ?*
  - *What will be the running time of the brute-force algorithm ?*

LCS Algorithm

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of x; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n\ 2^m)$
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems:
  – "find LCS of pairs of *prefixes* of X and Y"

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$
- $c[m,n]$ is the final solution.

Step 1: Define the problem with respect to subproblems

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Two cases: either the characters are the same or they're different

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$ A

LCS

The characters are part of the LCS

Y = $\boxed{\text{B D C A B}}$ A

If they're the same

$$LCS(X,Y) = LCS(X_{m-1}, Y_{n-1}) + 1$$

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$ B

LCS

Y = $\boxed{\text{B D C A B A}}$

If they're different

$$LCS(X, Y) = LCS(X_{m-1}, Y)$$

Step 1: Define the problem with respect to subproblems

$X = \boxed{A\ B\ C\ B\ D\ A\ B}$

LCS

$Y = \boxed{B\ D\ C\ A\ B}\ A$

If they're different

$$LCS(X,Y) = LCS(X, Y_{n-1})$$

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$ B

Y = $\boxed{\text{B D C A B A}}$

?

X = $\boxed{\text{A B C B D A B}}$

Y = $\boxed{\text{B D C A B}}$ A

If they're different

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{m-1}, Y_{n-1}) & \text{if } x_m = y_n \\ \max(LCS(X_{m-1}, Y), LCS(X, Y_{n-1})) & \text{otherwise} \end{cases}$$