Dynamic Programming

Introduction

Prof. Muhammad Saeed

Dynamic Programming

- Dynamic programming like the divide and conquer method, solves problem by combining the solutions of sub problems
- Divide and conquer method partitions the problem into independent sub problems, solves the sub problems recursively and then combine their solutions to solve the original problem.
- Dynamic programming is applicable, when the sub-problems are NOT independent, that is when sub-problems share sub sub-problems.
- It is making a set of choices to arrive at optimal solution.
- A dynamic programming algorithm solves every sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub-problem is encountered

Optimization Problems

- Dynamic problem is typically applied to Optimization Problems

- In optimization problems there can be many possible solutions. Each solution has a value and the task is to find the solution with the optimal (Maximum or Minimum) value. There can be several such solutions.

4 Steps of Dynamic Programming Algorithm

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution bottom-up.
- Construct an optimal solution from computed information

Often only the value of the optimal solution is required so step-4 is not necessary.

Dynamic Programming

- Dynamic programming relies on working "from the bottom up" and saving the results of solving simpler problems
  - These solutions to simpler problems are then used to compute the solution to more complex problems
- Dynamic programming solutions can often be quite complex and tricky
- Dynamic programming is used for optimization problems, especially ones that would otherwise take exponential time
  - Only problems that satisfy the principle of optimality are suitable for dynamic programming solutions
- Since exponential time is unacceptable for all but the smallest problems, dynamic programming is sometimes essential

Example: Binomial Coefficients

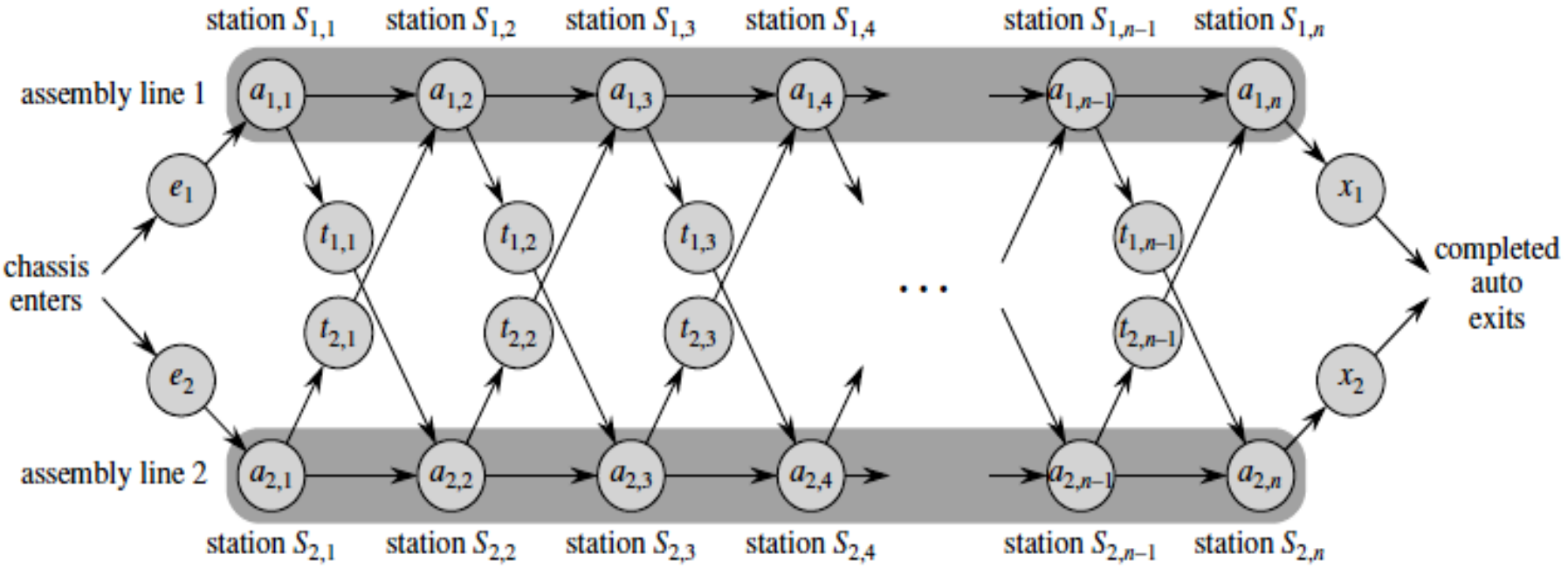
- $(x + y)^2 = x^2 + 2xy + y^2$, coefficients are 1,2,1
- $(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, coefficients are 1,3,3,1
- $(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, coefficients are 1,4,6,4,1
- $(x + y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$, coefficients are 1,5,10,10,5,1
- The $n+1$ coefficients can be computed for $(x + y)^n$ according to the formula $c(n, i) = n! / (i! * (n - i)!)$ for each of $i = 0..n$
- The repeated computation of all the factorials gets to be expensive
- We can use dynamic programming to save the factorials as we go

Solution by dynamic programming

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

- Each row depends only on the preceding row
- Only linear space and quadratic time are needed
- This algorithm is known as Pascal's Triangle

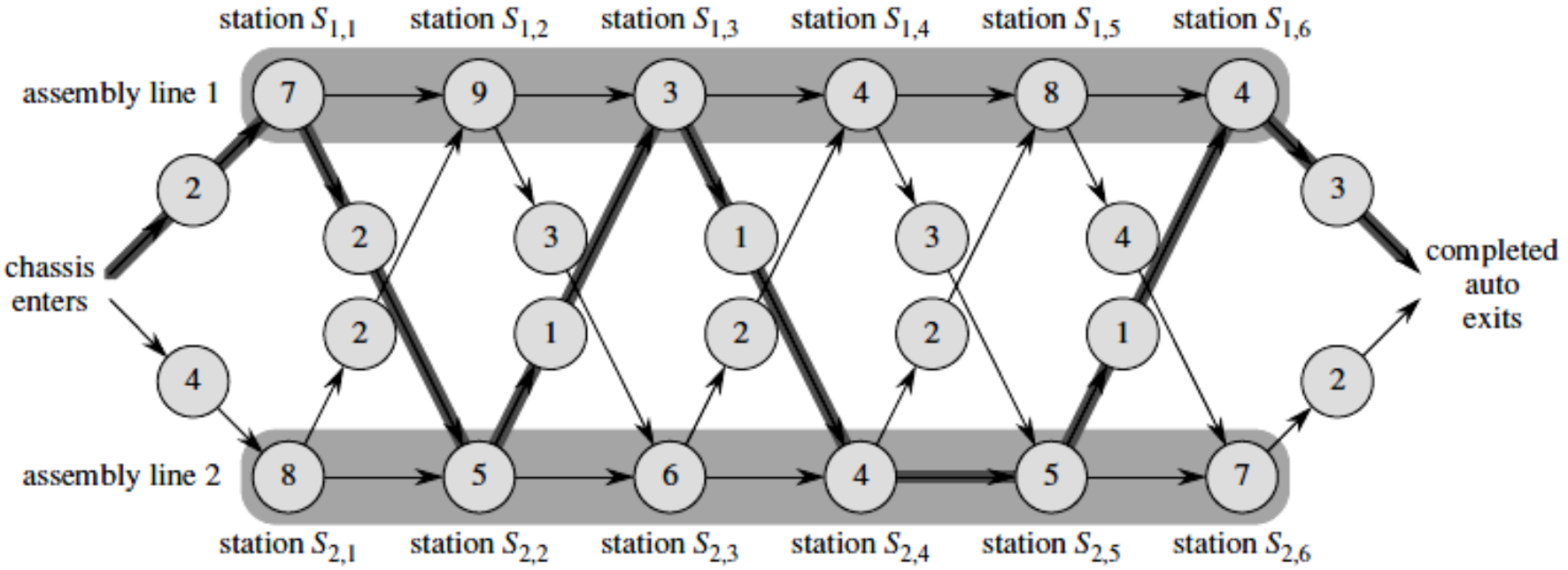
Assembly-line Scheduling .....



$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1\,, \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1\,, \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

….. Assembly-line Scheduling …..



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

….. Assembly-line Scheduling

```
FASTEST-WAY(a, t, e, x, n)
 1  f1[1] ← e1 + a1,1
 2  f2[1] ← e2 + a2,1
 3  for j ← 2 to n
 4       do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
 5            then f1[j] ← f1[j − 1] + a1,j
 6                 l1[j] ← 1
 7            else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
 8                 l1[j] ← 2
 9          if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
10            then f2[j] ← f2[j − 1] + a2,j
11                 l2[j] ← 2
12            else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
13                 l2[j] ← 1
14  if f1[n] + x1 ≤ f2[n] + x2
15     then f* = f1[n] + x1
16          l* = 1
17     else f* = f2[n] + x2
18          l* = 2
```

$$T(n) = \Theta(n)$$

Matrix-chain multiplication .....

- Matrix Chain-Product:
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- Example
  - B is 3 × 100
  - C is 100 × 5
  - D is 5 × 5
  - (B*C)*D takes 1500 + 75 = 1575 ops
  - B*(C*D) takes 1500 + 2500 = 4000 ops

….. Matrix-chain multiplication …..

A Greedy Approach

- Idea #1: repeatedly select the product that uses (up) the most operations.
- Counter-example:
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10
  - Greedy idea #1 gives (A*B)*(C*D), which takes 500+1000+500 = 2000 ops
  - A*((B*C)*D) takes 500+250+250 = 1000 ops

..... Matrix-chain multiplication .....

Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.
- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
- The greedy approach is not giving us the optimal value.

….. Matrix-chain multiplication …..

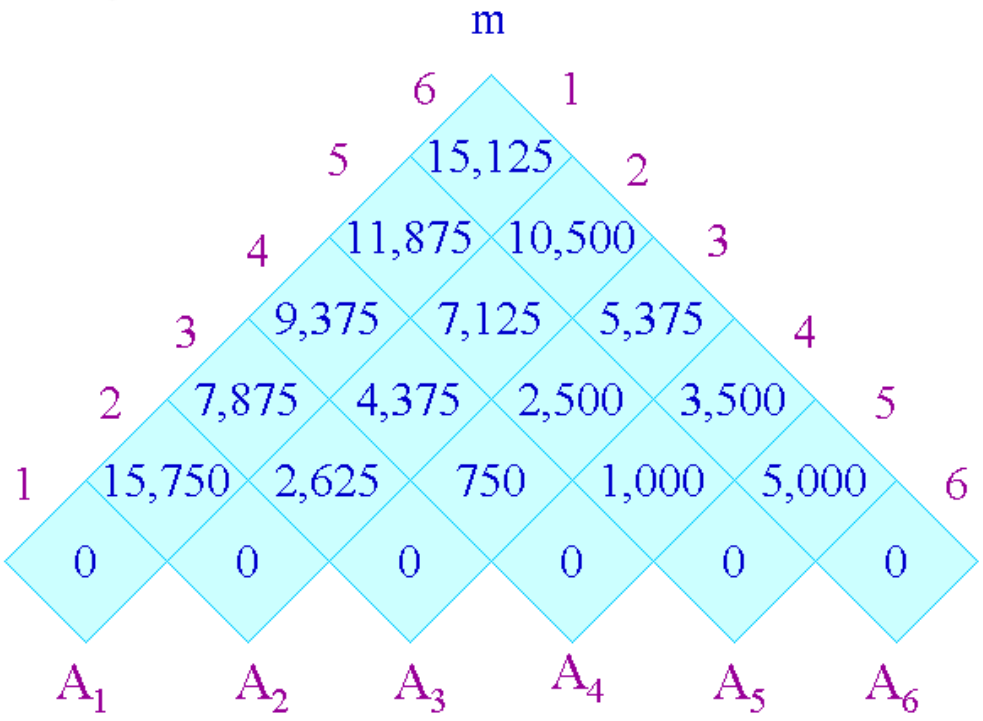
An Enumeration Approach

- Matrix Chain-Product Alg.:
  - Try all possible ways to parenthesize $A=A_0*A_1*…*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of paranthesizations is equal to the number of binary trees with n nodes
  - This is exponential!
  - It is called the Catalan number, and it is almost 4^n.

….. Matrix-chain multiplication …..

Matrix	Dimension
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

$$T(n) = O(n^3)$$



$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} = 7125.$$

….. Matrix-chain multiplication …..

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n          ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m and s
```

$$T(n) = O(n^3)$$

….. Matrix-chain multiplication

A "Recursive" Approach

- Define **subproblems**:
  - Find the best parenthesization of $A_i*A_{i+1}*…*A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- **Subproblem optimality**: The optimal solution can be defined in terms of optimal subproblems
  - There has to be a final multiplication (root of the expression tree) for the optimal solution.
  - Say, the final multiply is at index i: $(A_0*…*A_i)*(A_{i+1}*…*A_{n-1})$.
  - Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better "optimal" solution.

The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Fibonacci Numbers

Introduction

- Fibonacci numbers:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- The initial terms of the sequence
  - $(F_0, F_1, \ldots) = (0, 1, 1, 2, 3, 5, 8, 13, \ldots)$

..... Fibonacci Numbers .....

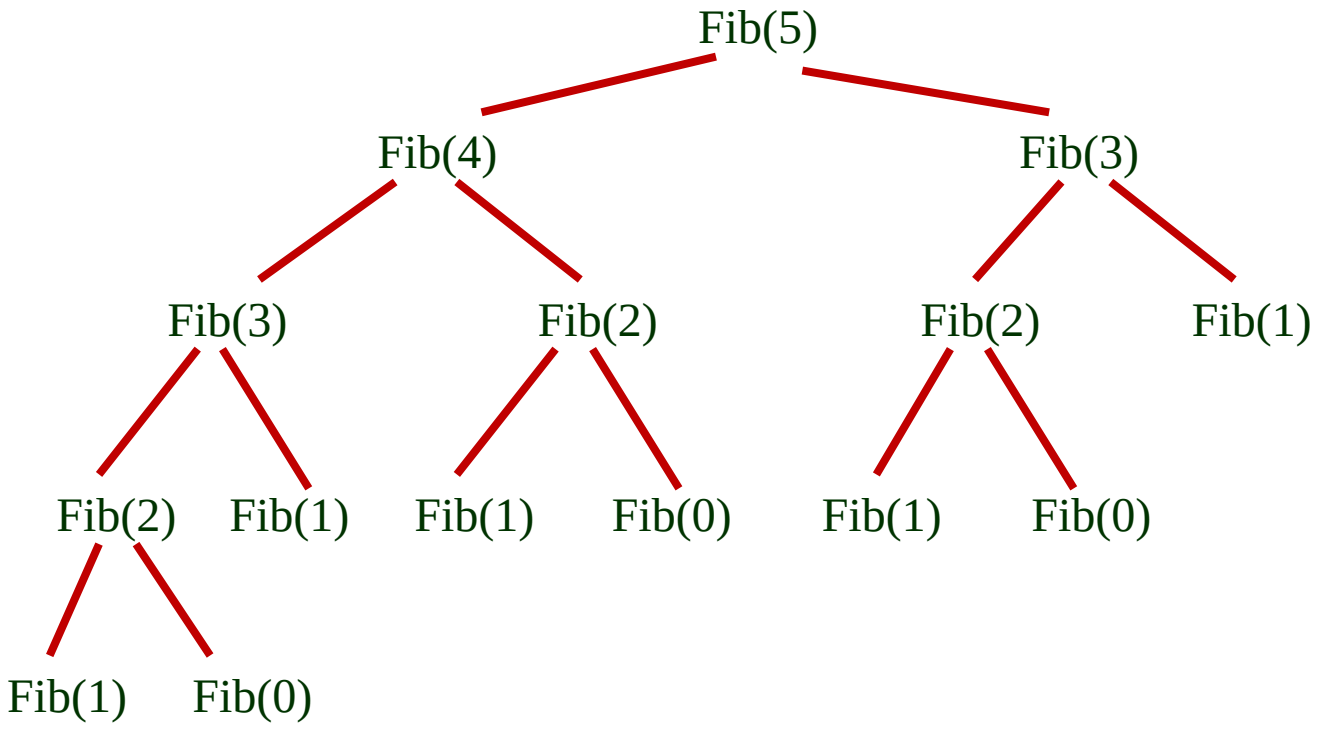
Computing Fibonacci Numbers

- There is an obvious (but terribly inefficient) recursive algorithm:
- void Fib(n)

```
{
    if (n == 0) or n==1 then
    return n;
 else
    return (F(n-1) + Fib(n-2))
 }
```

….. Fibonacci Numbers …..

Recursion Tree for Fib(5)



..... Fibonacci Numbers .....

Number of Recursive Calls

- The leafs of the recursion tree have values Fib(0)=0 or Fib(1)=1.
- Since Fib(n) can be calculated as the sum of all values in the leafs, there must be Fib(n) leafs with the value 1.
- This approach repeats unnecessary calculations
- Employing Dynamic Programming technique last calculated values are stored in a table to access it in next step.

..... Fibonacci Numbers .....

No Recursion

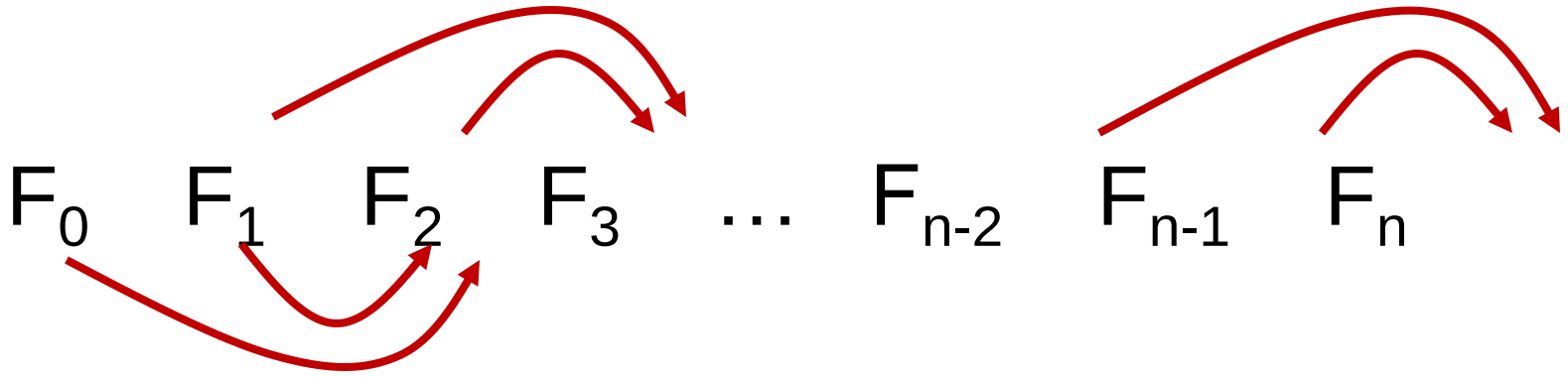
- Recursion adds overhead
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- Avoid the recursion overhead by filling in the table entries bottom up, instead of top down.

..... Fibonacci Numbers .....

Subproblem Dependencies

- Figure out which subproblems rely on which other subproblems

- Example:



..... Fibonacci Numbers .....

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

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

..... Fibonacci Numbers .....

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

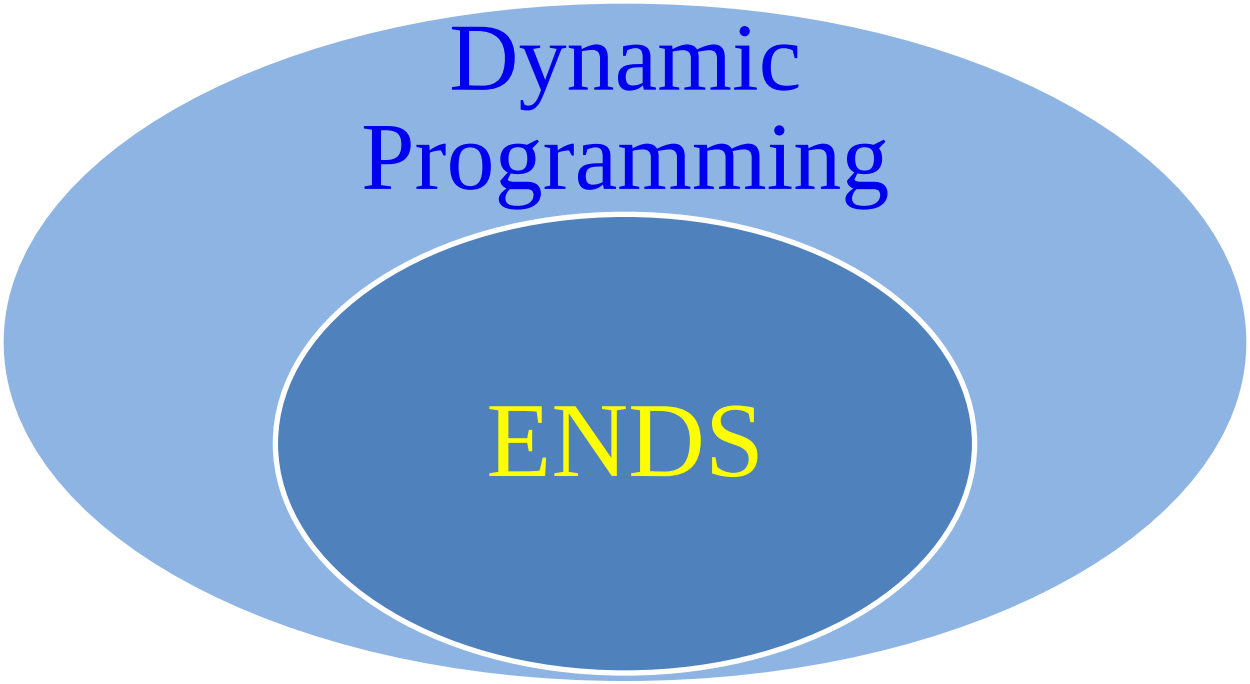
e.g., save space by only keeping last two numbers computed

..... Fibonacci Numbers .....

More Efficient Recursive Algorithm

- 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])

- Computes each F[i] only once.
- This technique is called **memoization**


Dynamic
Programming
ENDS