Ch18: Dynamic Programming

305234

Algorithm Analysis and Design

Jiraporn Pooksook
Naresuan University

Dynamic Programming

- Solves problems by combing the solutions to subproblems.
- Similar to divide-and –conquer method but dynamic programming is applicable when subproblems are not independent, that is , when subproblems share subsubproblems.
- A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

Dynamic Programming

- Dynamic programming is typically applied to optimization problems, can be many possible solutions. We wish to find a solution with the optimal (minimum or maximum) value.

Fibonacci Numbers

- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}$
- Goal : to compute F_n

Naïve Recursive Algorithm

```
fib(n):
        if n <= 2
                then f = 1
        else    f = fib(n-1) + fib(n-2)
return f
```

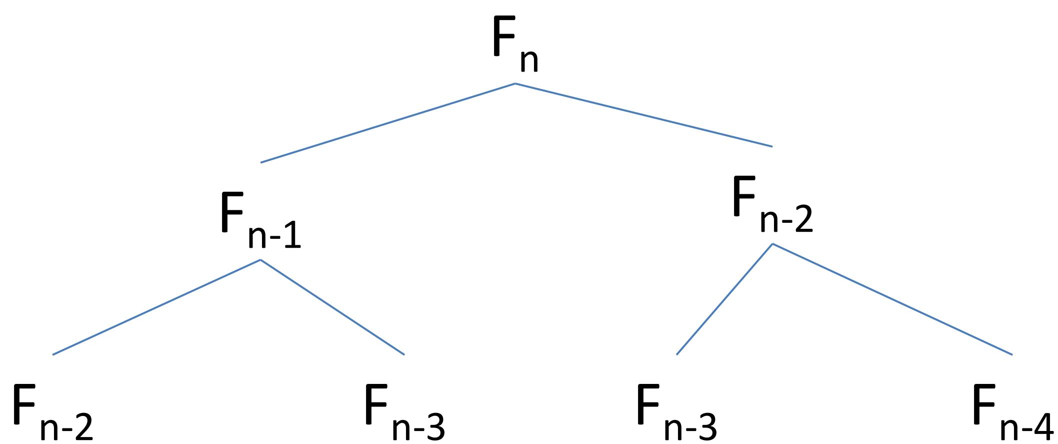
Exponential running time !!

T(n) = T(n-1) + T(n-2) + $\Theta(1)$

>= 2 T(n-2)

= $\Theta(2^{n/2})$

Naïve Recursive Algorithm

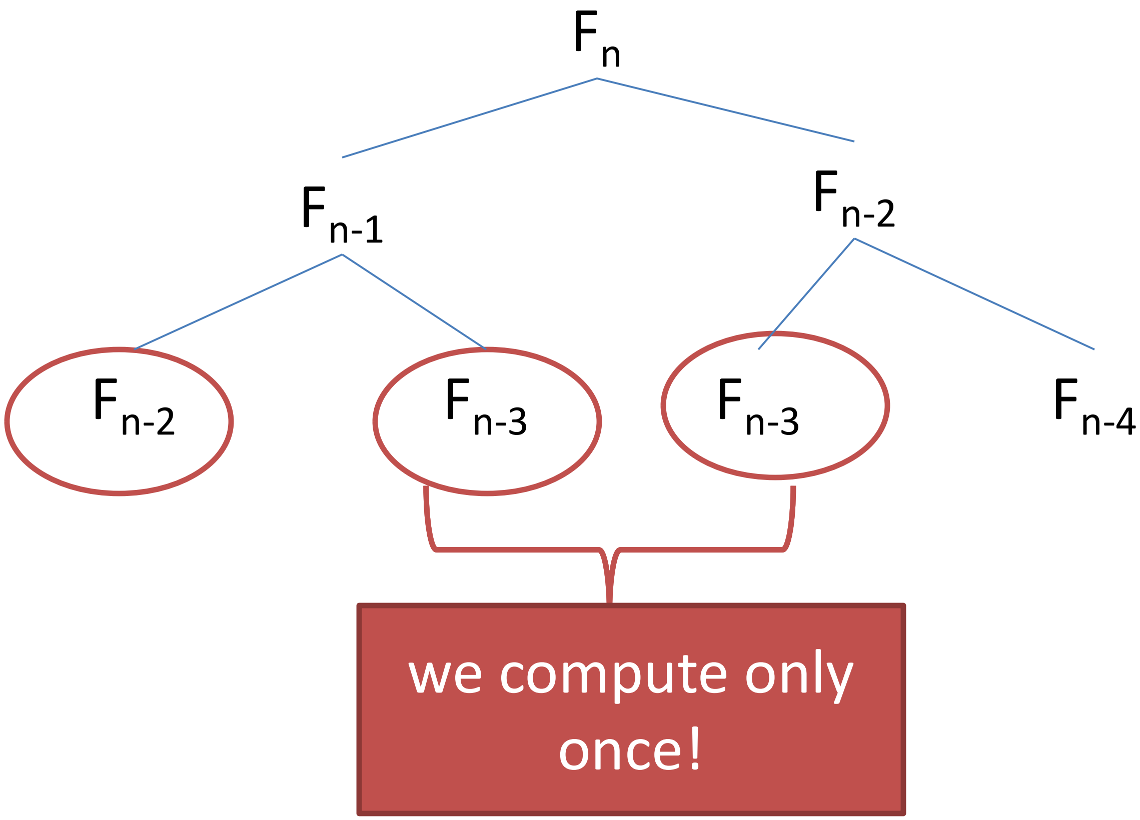


Memoized Dynamic Programming Algorithm

```
memo= { }
fib(n) :
        if n is in memo
                then return memo[n]
        if n <=2
                then f =1
        else    f = fib(n-1) + fib(n-2)

        memo[n] = f
return f
```

Memoized Dynamic Programming Algorithm



Memoized Dynamic Programming Algorithm

- fib(k) only recurses the first time it is called.
- For all k, memoized calls cost $\Theta(1)$
- The number of nonmemoized call is n

  fib(1), fib(2), … , fib(n)
- The non-recursive work per call is $\Theta(1)$
- Hence running time = $\Theta(n)$

Dynamic Programming

- Dynamic programming algorithm in general is to memorize and re-use solutions to subproblems that help solving the problem.
- Hence dynamic programming is a recursion and memoization.
- The running time is equal to the number of subproblems x (time/subproblem)
  - Ex: $n \times \Theta(1) = \Theta(n)$

Don't count memoized recursion!!

Bottom-up Dynamic Programming algorithm

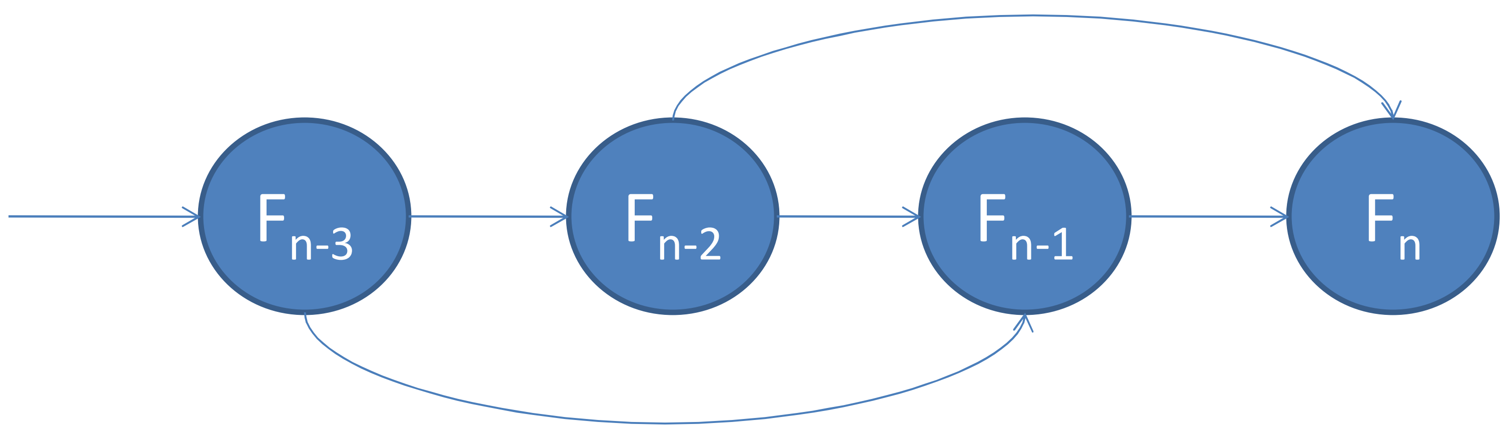
```
fib= { }
for  k from 1 to n :
        if k <=2
                then f =1
        else    f = fib[k-1] + fib[k-2]

        fib[k] = f
return fib[n]
```

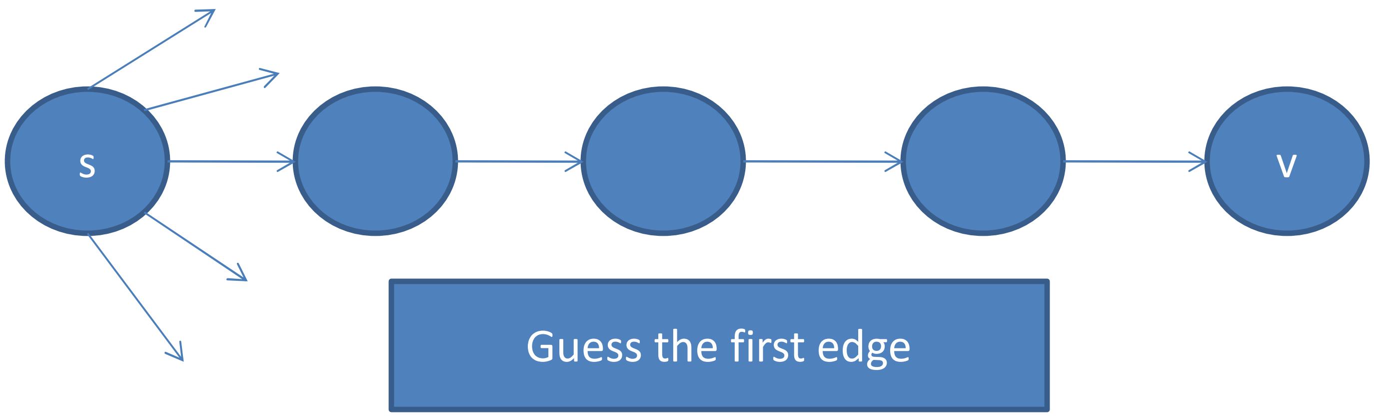
Running time is Θ(n)

Bottom-up Dynamic Programming algorithm

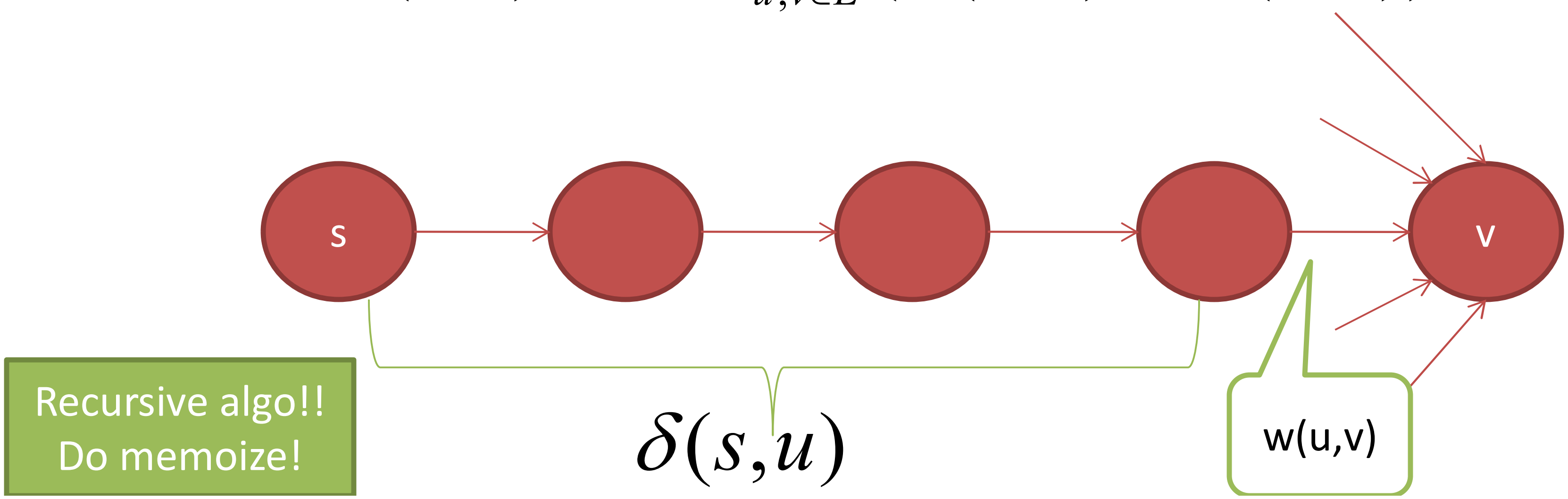
- It has exactly the same computation to memoization.
- It uses topological sort of subproblems dependency.
  - DAG



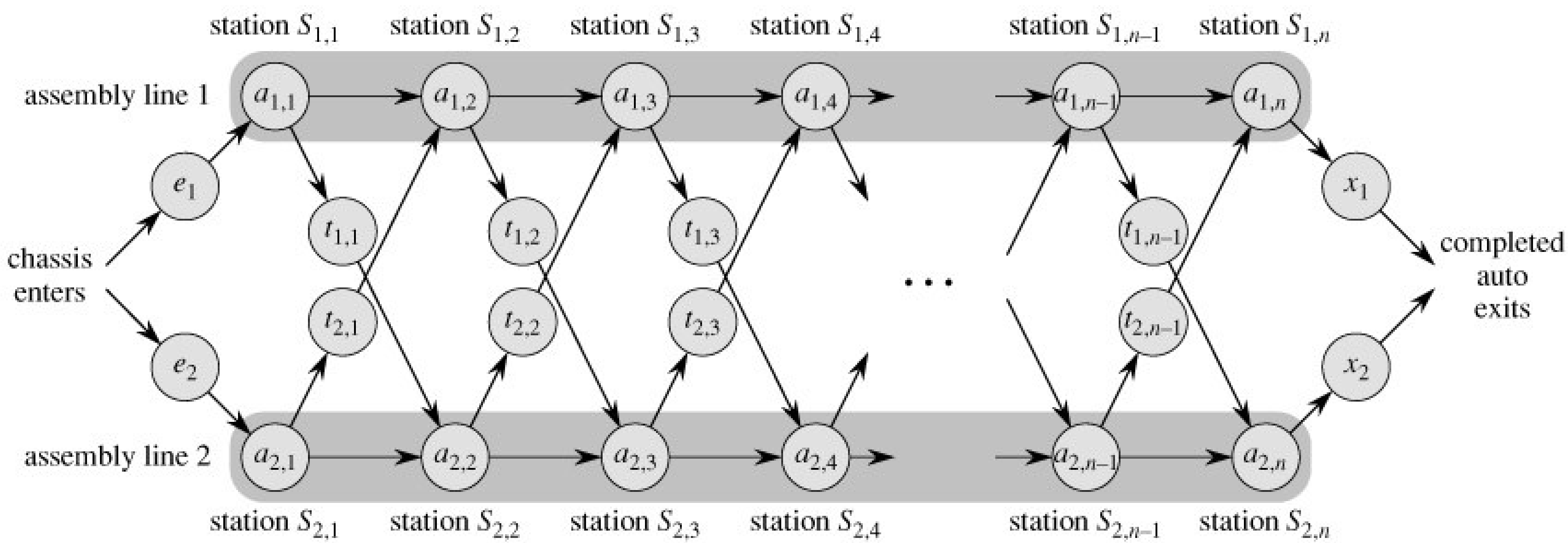
Single-Source Shortest Paths



$$\delta(s,v) = \min_{u,v \in E}(\delta(s,u) + w(u,v))$$



Assembly-line Scheduling



Assembly-line Scheduling

- The structure of the fastest way through the factory.
- There are 2 choices:
- Come from station $S_{1,j-1}$ and then directly to station $S_{1,j}$
- Come from station $S_{2,j-1}$ and then been transferred to station $S_{1,j}$

Assembly-line Scheduling

- A recursive solution
- The fastest time to get a chassis all the way through the factory is denoted by f*.
- $f^* = \min(f_1[n] + x_1 , f_2[n] + x_2)$
- $f_1[1] = e_1 + a_{1,1}$
- $f_2[1] = e_2 + a_{2,1}$

Assembly-line Scheduling

- A recursive solution
- $f_1[j] = f_1[j-1] + a_{1,j}$, and
- $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$
- $f_1[j] = \min (f_1[j-1] + a_{1,j} , f_2[j-1] + t_{2,j-1} + a_{1,j})$

- $f_2[j] = f_2[j-1] + a_{2,j}$, and
- $f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}$
- $f_2[j] = \min (f_2[j-1] + a_{2,j} , f_1[j-1] + t_{1,j-1} + a_{2,j})$

Assembly-line Scheduling

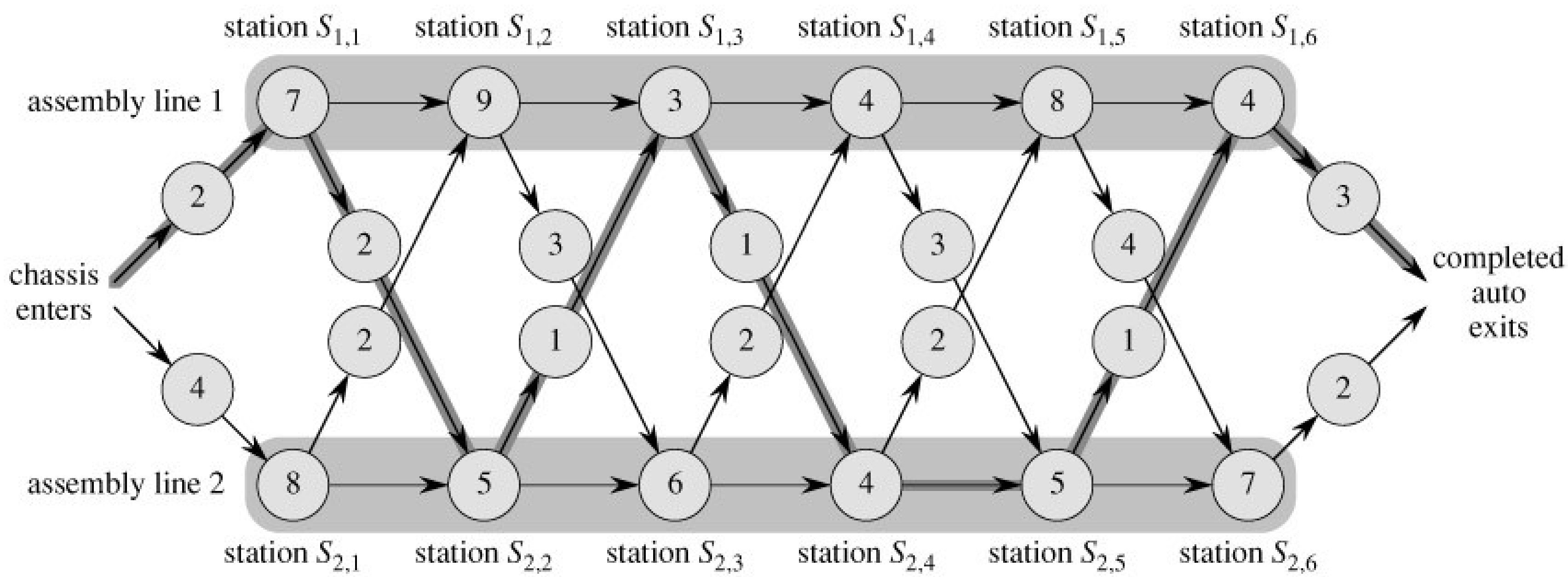
- A recursive solution
- $f_1[j] = e_1 + a_{1,1}$ if $j = 1$
- $f_1[j] = \min(f_1[j-1] + a_{1,j}\ ,\ f_2[j-1] + t_{2,j-1} + a_{1,j})$

  if $j \geq 2$

- $f_2[1] = e_2 + a_{2,1}$ if $j=1$
- $f_2[j] = \min(f_2[j-1] + a_{2,j}\ ,\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

  if $j \geq 2$

Assembly-line Scheduling



(a)

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

$l^* = 1$

(b)

Assembly-line Scheduling

- Computing the fastest times
- If we use recursion, the running time will be
- $\Theta(2^{n/2})$
- If we use bottom-up dynamic programming, the running time will be only Θ(n)

Fastest-Way(a, t, e, x, n)

```
f_1[1] = e_1 + a_{1,1}
f_2[1] = e_2 + a_{2,1}
for j = 2 to n
        do if f_1[j-1] + a_{1,j}  ≤ f_2[j-1] + t_{2,j-1} + a_{1,j}
             then f_1[j] = f_1[j-1] + a_{1,j}
                      l_1[j] = 1
             else f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}
                      l_1[j] = 2

           if f_2[j-1] + a_{2,j}  ≤ f_1[j-1] + t_{1,j-1} + a_{2,j}
             then f_2[j] = f_2[j-1] + a_{2,j}
                      l_1[j] = 2
             else f_2[j] = f_1[j-1] + t_{1,j-1} + a_{2,j}
                      l_1[j] = 1
if f_1[n] + x_1 ≤ f_2[n] + x_2
        then f* = f_1[n] + x_1
               l* = 1
        else   f* = f_2[n] + x_2
               l* = 2
```

Print-Stations(l , l*, n)

```
i = l*
print "line" i",sation "n
for  j = n downto 2
        do i =  l_i[j]
        print "line" i",station" j-1
```