COMP 6651 / Winter 2022 - B. Jaumard

Lecture on Dynamic Programming

February 4th, 2022

Outline

Dynamic Programming: Dependent Subproblems

- An algorithm design technique (like divide and conquer).
- Divide-and-conquer techniques, e.g., quick-sort.
  - Partition the problem into independent subproblems.
  - Solve the subproblems recursively.
  - Combine the solution to solve the original problem
- Dynamic programming applicable when subproblems are not independent, subproblems share information/subsubproblems.

In the context of dependent subproblems :

- A divide-and-conquer algorithm does extra work, repeatedly solving common subsubproblems,
- Dynamic programming algorithm solves every subsubproblem just once and then saves its answer in a table, avoiding extra work of recomputing when the subsubproblem is encountered.

Dynamic Programming: Optimization Problems

- Dynamic Programming is applied to optimization problems.
- An optimization problem: an objective + a set of constraints.
- Many possible solutions: we wish to find a solution with the optimal value of the objective (max or min).
- An optimal solution as opposed to the optimal value: Several solutions can achieve the optimal value.

Dynamic Programming: Four Steps

The development of a dynamic programming algorithm can be broken into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

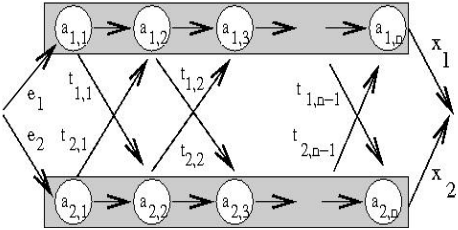
Four Illustrations of Dynamic Programming

- Assembly-line scheduling
- Knapsack problem
- Shortest path problem
- Longest common subsequence

Assembly-Line Scheduling Problem

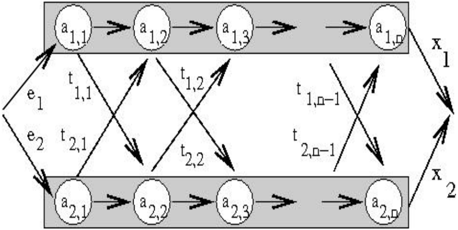
Definition of the Assembly-Line Scheduling Problem (1/3)

- Automobile factory with two assembly lines
  - Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Entry times e_1 and e_2 and exit times x_1 and x_2



Definition of the Assembly-Line Scheduling Problem (2/3)

- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, $j = 1, \ldots, n-1$

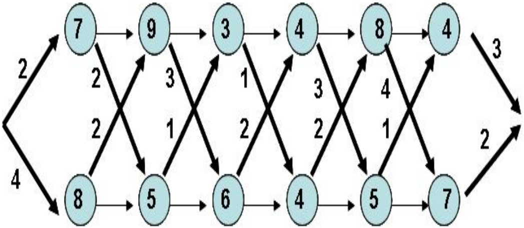


The Assembly-Line Scheduling Problem

Assembly-Line Scheduling Problem

Determine which stations to choose from line 1 and which to choose from line 2 in order to **minimize** the total time through the factory for one auto.

- Stations 1, 3 and 6 from line 1
- Stations 2, 4, and 5 from line 2
- Cost: 38

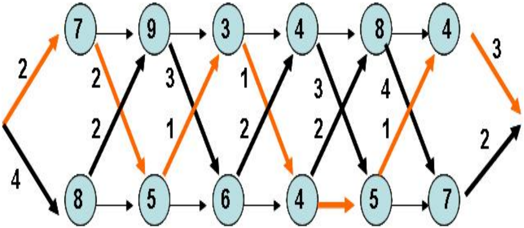


The Assembly-Line Scheduling Problem

Assembly-Line Scheduling Problem

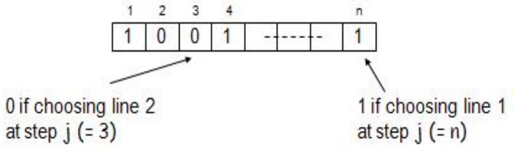
Determine which stations to choose from line 1 and which to choose from line 2 in order to **minimize** the total time through the factory for one auto.

- Stations 1, 3 and 6 from line 1
- Stations 2, 4, and 5 from line 2
- Cost: 38



Brute Force Approach

- Brute force
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one
- Problem



  - There are 2^n possible ways to choose stations
  - Infeasible when n is large

Dynamic Programming: Step 1 (1/2)

Step 1 : Characterize the structure of an optimal solution

- Fastest possible way for a chassis to get from the starting point through $S_{i,j}$, (*for* $j = 1$) only one way.
- For $j = 2, ..., n$ two choices;
  - $S_{1,j-1} \hookrightarrow S_{1,j}$ or
  - $S_{2,j-1} \hookrightarrow S_{1,j}$, with a transfer time from line 2 to 1 of $t_{2,j-1}$.

Dynamic Programming: Step 1 (2/2)

Step 1 : Characterize the structure of an optimal solution

Optimal substructure property:

An optimal solution to a problem (Finding the fastest way through station $S_{i,j}$) contains within it an optimal solution to subproblems (finding the fastest way through either $S_{1,j-1}$ or $S_{2,j-1}$).

Thus the fastest way through station $S_{1,j}$ is either:

- the fastest way through station $S_{1,j-1}$ and then directly through station $S_{1,j}$, or
- the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.

Symmetric reasoning for the fastest way through station $S_{2,j}$

Dynamic Programming: Step 2 (1/4)

Step 2: Recursively define the value of an optimal solution in terms of the optimal solutions to subproblems.

- $f_i[j]$: fastest possible time to get a chassis from the starting point through station $S_{i,j}$
- f^*: optimal value $\hookrightarrow$ fastest time to get a chassis all the way through the factory.

$$f^* = \min\{f_1[n] + x_1, f_2[n] + x_2\}$$

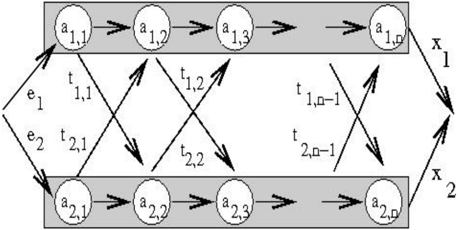
Dynamic Programming: Step 2 (2/4)

- Chassis has to get all the way through station n on either line 1 or 2, and then to the factory exit. Since the fastest of these ways is the fastest way throughout the entire factory, we have

$$f^* = \min\{f_1[n] + x_1, f_2[n] + x_2\}$$

- Also easy to reason about $f_1[1]$ and $f_2[1]$: To go through station 1 on either line, a chassis just goes directly to that station:

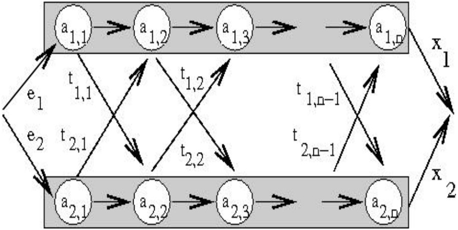
$$f_1[1] = e_1 + a_{1,1} \qquad f_2[1] = e_2 + a_{2,1}$$



Dynamic Programming: Step 2 (3/4)

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

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



Dynamic Programming: Step 2 (4/4)

- $f_i[j]$: values of the optimal solutions of the subproblems.
- In order to keep track of how the optimal solution is constructed: $\ell_i[j]$ = line number, 1 or 2, whose **station** $j-1$ is used in a fastest way through station $S_{i,j}$.
- No need to define $\ell_i[1]$: no station precedes station 1 on either line.
- ℓ^*: line whose station n is used in a fastest way through the entire factory.

Dynamic Programming: Step 3 (1/3)

Step 3. Compute the value of an optimal solution in a bottom-up fashion

- We can compute the fastest way through the factory and the time it takes in $\Theta(n)$ time, by computing $f_i[j]$ in order of increasing stations j
- The FASTEST-WAY procedure takes as input the values $a_{i,j}$, $t_{i,j}$, e_i, and x_i , as well as n, the number of stations in each assembly line.

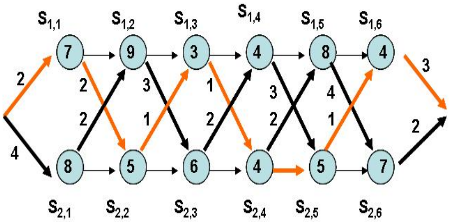
Dynamic Programming: Step 3 (2/3)

FastestWay(a, t, e, x, n)

$f[1,1] = e[1] + a[1,1];$
$f[2,1] = e[2] + a[2,1];$
for $(j = 2; j \leq n, j++)$
  if $(f[1,j-1] + a[1,j] \leq f[2,j-1] + t[2,j-1] + a[1,j])$
    $\{f[1,j] \leftarrow f[1,j-1] + a[1,j]; l[1,j] \leftarrow 1;\}$
  else $\{f[1,j] \leftarrow f[2,j-1] + t[2,j-1] + a[1,j]; \ell[1,j] \leftarrow 2;\}$ **(end elseIf)**
  if $(f[2,j-1] + a[2,j] \leq f[1,j-1] + t[1,j-1] + a[2,j])$
    $\{f[2,j] \leftarrow f[2,j-1] + a[2,j]; \ell[2,j] \leftarrow 2;\}$
  else $\{f[2,j] \leftarrow f[1,j-1] + t[1,j-1] + a[2,j]; \ell[2,j] \leftarrow 1;\}$ **(end elseIf)**
end for

if $(f[1,n] + x[1] \leq f[2,n] + x[2])$
  $\{f^* \leftarrow f[1,n] + x[1]; \ell^* \leftarrow 1;\}$
else
  $\{f^* \leftarrow f[2,n] + x[2]; \ell^* \leftarrow 2;\}$
end elseIf

Dynamic Programming: Step 3 (3/3)



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
$\ell_1[j]$	1	2	1	1	2
$\ell_2[j]$	1	2	1	2	2

$$\ell^* = 1$$

Dynamic Programming: Step 4 (1/2)

Step 4 : Construct an optimal solution from computer solution, i.e., Constructing the fastest way through the factory. It is done from computed values $f_i[j]$, f^*, $\ell_i[j]$, ℓ^*.

PRINT-STATIONS(l, l^*, n)

1 $i \leftarrow \ell^*$
2 print line "i", station "n"
3 for $j \leftarrow n$ **downto** 2
4 **do** $i \leftarrow \ell_i[j]$
5 print line "i", station "$j - 1$"

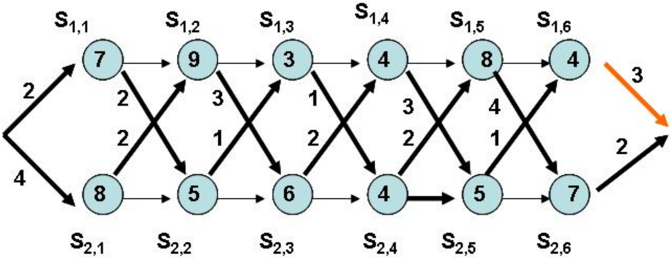
j	2	3	4	5	6
$\ell_1[j]$	1	2	1	1	2
$\ell_2[j]$	1	2	1	2	2

$$\ell^* = 1$$

line 1, station 6
line 2, station 5
line 2, station 4
line 1, station 3
line 2, station 2
line 1, station 1

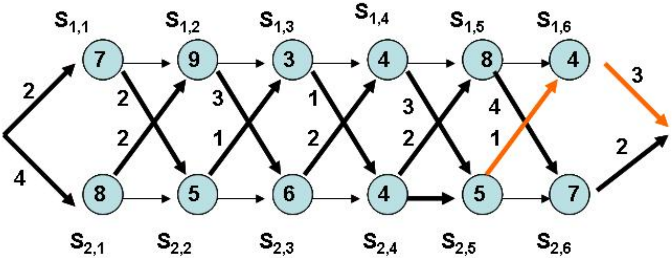
Dynamic Programming: Step 4 (2/2)

$\ell^* = 1$



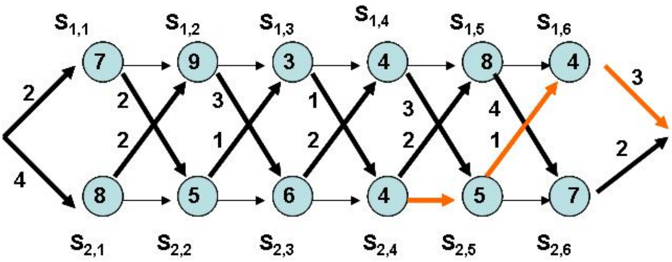
Dynamic Programming: Step 4 (2/2)

$\ell_1[6] = 2$



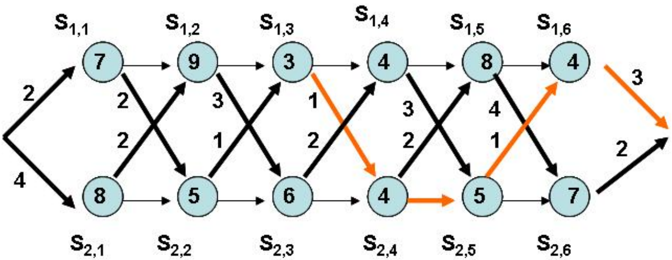
Dynamic Programming: Step 4 (2/2)

$\ell_2[5] = 2$



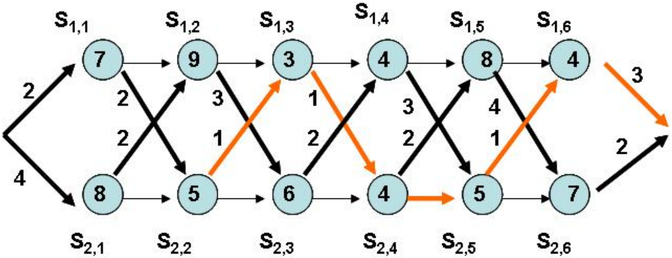
Dynamic Programming: Step 4 (2/2)

$\ell_2[4] = 1$



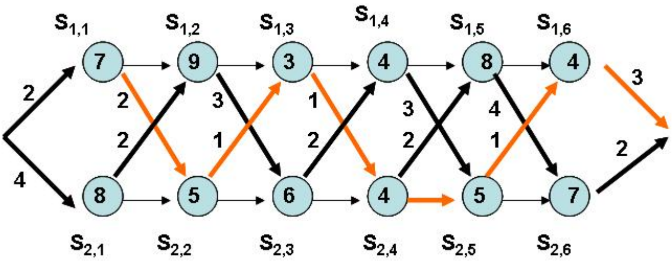
Dynamic Programming: Step 4 (2/2)

$\ell_1[3] = 2$



Dynamic Programming: Step 4 (2/2)

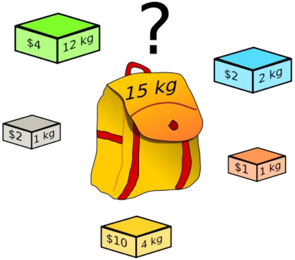
$\ell_2[2] = 1$



Knapsack Problem

The 0/1 Knapsack Problem (1/2)

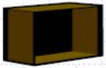
- Given: A set S of n items, with each item i having
  - w_i a positive weight
  - b_i a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.



Items:

	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

"knapsack"



box of width 9 in

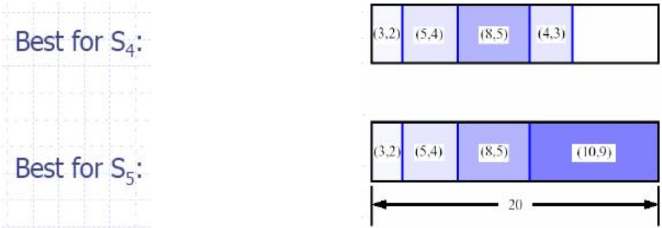
Solution:
- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)

The 0/1 Knapsack Problem (2/2)

- Given: A set S of n items, with each item i having
  - w_i a positive weight
  - b_i a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- Well known problem, usually expressed as a mathematical program
  - Objective: maximize $\sum_{i \in T} b_i x_i$
  - Constraint: $\sum_{i \in T} w_i x_i \leq W$
    - T: index set of the selected items
    - $x_i \in \{0, 1\}, i \in T$ (decision variables)
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.

Solving the 0/1 Knapsack Problem: First Attempt

- S_k: Set of items numbered 1 to k.
- Define $B[k]$ = best selection from S_k.
- Difficulty: does not have subproblem optimality:
  - Consider set $S = \{(3,2),(5,4),(8,5),(4,3),(10,9)\}$ of (benefit, weight) pairs and total weight $W = 20$



Solving the 0/1 Knapsack Problem: Second Attempt

- S_k: Set of items numbered 1 to k.
- Define $B[k, w]$ to be the value of the best selection from S_k with weight at most w
- Good news: $B[k, w]$ has the subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k] + b_k\} & \text{otherwise.} \end{cases}$$

i.e., the best subset of S_k with weight at most w is either

- the best subset of S_{k-1} with weight at most w or
- the best subset of S_{k-1} with weight at most $w - w_k$ plus item k

Solving the 0/1 Knapsack Problem using Dynamic Programming (1/2)

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{otherwise.} \end{cases}$$

- Running time: $O(nW)$.
- Not a polynomial-time algorithm since W may be large
- This is a **pseudo-polynomial** (running time is polynomial in the numerical value of the input, but exponential in the length of the input) time algorithm:

Solving the 0/1 Knapsack Problem using Dynamic Programming (2/2)

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{otherwise.} \end{cases}$$

$B[k,w]$ is defined in terms of $B[k-1,*]$
$\hookrightarrow$ use of **only two one-dimensional arrays** of length $W+1$.
For each k, A stores $B[k-1,*]$ values and $A+$ the $B[k,*]$ ones.

- **Input:** Set S of n items with benefit b_i and weight w_i; maximum weight W
- **Output:** Benefit of best subset of S with weight at most W

Space requirement: $O(W)$

Knapsack Problem

for $w \leftarrow 0$ **to** W **do**
  $A^+[w] \leftarrow 0$
for $k \leftarrow 1$ **to** n **do**
  copy array A^+ into array A
  for $w \leftarrow w_k$ **to** W **do**
    if $A[w-w_k]+b_k > A[w]$ **then**
      $A^+[w] \leftarrow A[w-w_k]+b_k$
return $A^+[w]$

0-1 Knapsack: An Example (1/7)

- 4 items
- Benefit values
  - $b_1 = 3$; $b_2 = 4$; $b_3 = 5$; $b_4 = 6$
- Weight values
  - $w_1 = 2$; $w_2 = 3$; $w_3 = 4$; $w_4 = 5$
- Objective: maximize the benefit while not exceeding the knapsack capacity ($W = 5$)

0-1 Knapsack: An Example (2/7)

k/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

Base case: for $w = 0$ to W: $B[0, w] = 0$

0-1 Knapsack: An Example (3/7)

k/w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Base case: for $w = 0$ to W: $B[0, w] = 0$
for $i = 1$ to n: $B[i, 0] = 0$

0-1 Knapsack: An Example (4/7)

k/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
Pointers		(0,1)	(0,0)	(0,1)	(0,2)	(0,3)
2	0					
3	0					
4	0					

$$B[1,w] = \begin{cases} B[0,w] = 0 & \text{if } w_1 = 2 > w \rightsquigarrow w \leq 1 \\ \max\{B[0,w], B[0,w-w_1] + c_1\} & \\ \quad = B[0,w-2] + 3 & \text{if } w \geq 2 \end{cases}$$

0-1 Knapsack: An Example (5/7)

k/w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
Pointers		(0,1)	(0,0)	(0,1)	(0,2)	(0,3)
2	0	0	3	4	4	7
Pointers		(1,1)	(1,2)	(1,0)	(1,1)	(1,2)
3	0					
4	0					

$$B[2, w] = \begin{cases} B[1, w] & \text{if } w_2 = 3 > w \rightsquigarrow w \leq 2 \\ \max\{B[1, w - w_2] + c_2, B[1, w]\} & \\ \quad = \max\{B[1, w - 3] + 4, B[1, w]\} & \\ \quad = 4 & \text{if } w = 3 \text{ or } 4 \\ \quad = 7 & \text{if } w \geq 5 \end{cases}$$

0-1 Knapsack: An Example (6/7)

k/w	0	1	2	3	4	5
0	**0**	$\leftarrow$ 0	0	0	0	0
1	0	0	$\nwarrow$ **3**	$\leftarrow$ 3	$\leftarrow$ 3	3
2	0	0	3	4	4	$\nwarrow$ 7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Optimal Value: 7

How to get the optimal solution: Use the pointers ...

0-1 Knapsack: An Example (7/7)

k/w	0	1	2	3	4	5	
0	0	0	0	0	0	0	
1	0	0	3	3	3	3	Take item #1
2	0	0	3	4	4	7	Take item #2
3	0	0	3	4	5	7	Do not take item #3
4	0	0	3	4	5	7	Do not take item #4

Optimal Value: 7

Optimal solution: (1,1,0,0)

How to retrieve the optimal vector? Use the pointers (not represented here). Start with $B(4,5)$. It was obtained using value $B(3,5)$:

$$B(4,5) = \max \left\{ \underbrace{B(3,5),}_{\text{Do not take item \#4}} \quad \underbrace{B(2, 5 - w_4) + c_4}_{\text{Take item \$4}} \right\} = B(3,5),$$

which is turn was obtained using value $B(2,5)$. And so on.

Shortest Path(s)

Dijkstra's Algorithm

- Dijkstra's algorithm finds shortest paths along certain types of graphs (acyclic directed graphs).
- It also belongs to the Dynamic Programming family, and, as such, its logic rests within the optimality criterion;
- Solve problems efficiently by overlapping subproblems and optimal substructure.

Dijkstra's Algorithm (from Lecture on Graph Algorithms)

- S: set of vertices whose final shortest-path weights from the source s have already been determined
- Algorithm selects the vertex $u \in V \setminus S$ with the shortest-path estimate, adds u to S, and relaxes all edges leaving u.
- Use a min-priority queue Q of vertices in $V \setminus S$, keyed by their d values.

Dijkstra, p.595 Cormen

```
Initialize-Single-Source(G,s)
S ← ∅
Q ← V
while Q ≠ ∅
   do u ← Extract-Min(Q)
      S ← S ∪ {u}
      for each vertex v ∈ ADJ[u]
         do Relax(u, v, w)
```

Dijkstra's Algorithm: A Dynamic Programming Approach

- The algorithm works from a source (s) by computing for each vertex v pertaining to V the cost $d[v]$ of the shortest path found so far between s and v.
- Initially,
  - $d[s] = 0$ for the source,
  - $d[v] = +\infty$ for every v in $V \setminus \{s\}$.
- When the algorithm finishes, $d[v]$ is the cost of the shortest path from s to v (or infinity, if no path exists).

Dijkstra's Algorithm: A Dynamic Programming Approach - Cont'd

Initialization:

$d[s] = 0$ for the source node

$d[v] = +\infty$ for every v in $V \setminus \{s\}$

Q (priority queue) $\leftarrow V$

$u \leftarrow s$

Iteration:

While ($|Q| > 1$ and $d[u] < +\infty$) Do:

  Update Q: $Q \leftarrow Q \setminus \{u\}$

  Update $d[v]$ for $v \in \omega^{+}(u) \cap Q$:

$$d[v] \leftarrow \min\{d[v], d[u] + \text{DIST}(u, v)\}$$

  Update u: $u \leftarrow \arg\min_{v \in Q} d[v]$

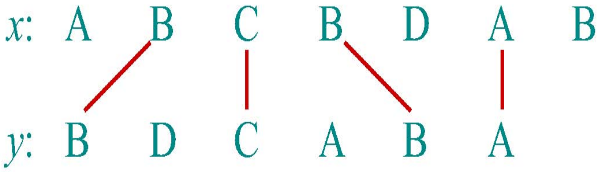
Longest Common Subsequence

Dynamic Programming

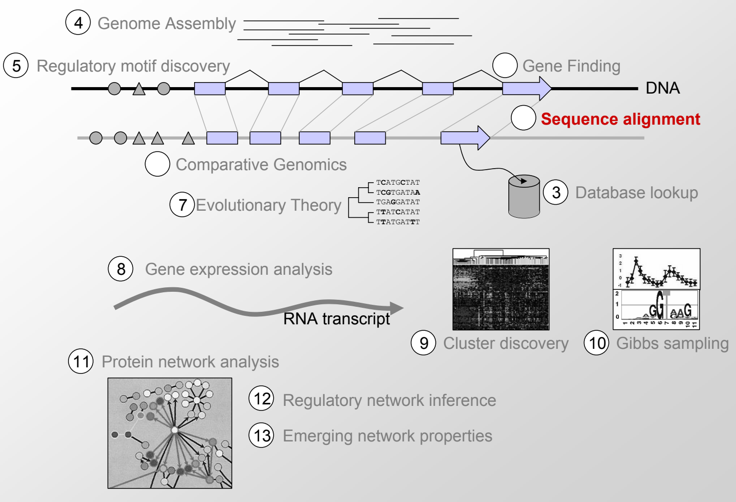
Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $\mathbf{x}[1, 2, \ldots, \mathbf{m}]$ and $\mathbf{y}[1, 2, \ldots, \mathbf{n}]$, find a longest subsequence common to both of them.
- “a” LCS, not “the” LCS
- LCS(x, y) = BCBA



Sequence Alignment in Computational Biology



Brute-force LCS algorithm

Check every subsequence of $x[1, 2, \ldots, m]$ to see if it is also a subsequence of $y[1, 2, \ldots, n]$.

Analysis

- Checking = $O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$ = exponential time.

Towards a better algorithm

Simplification:

1. Look at the length of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

Notation: Denote the length of a sequence s by $|s|$.

Strategy: Consider prefixes of x and y.

- Define $c[i, j] = |\mathrm{LCS}(x[1, 2, \ldots, i], y[1, 2, \ldots, j])|$.
- Then, $c[m, n] = |\mathrm{LCS}(x, y)|$.

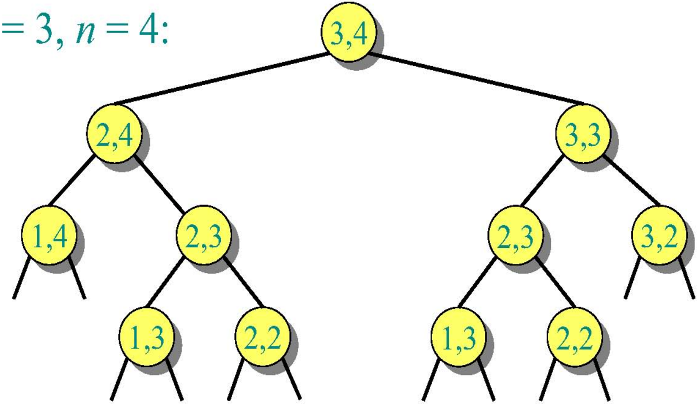
Recursive Algorithm for LCS

LCS(x, y, i, j)
if $x[i] = y[j]$
then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), LCS(x, y, i, j-1)\}$

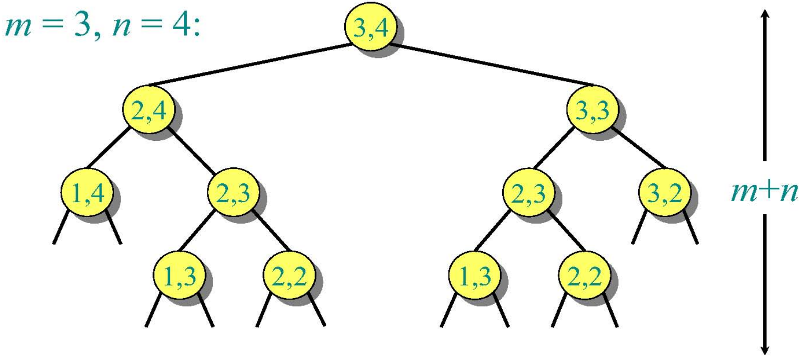
Worst Case: $x[i] \neq y[j]$, in which case, the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion Tree

$m = 3, n = 4$:

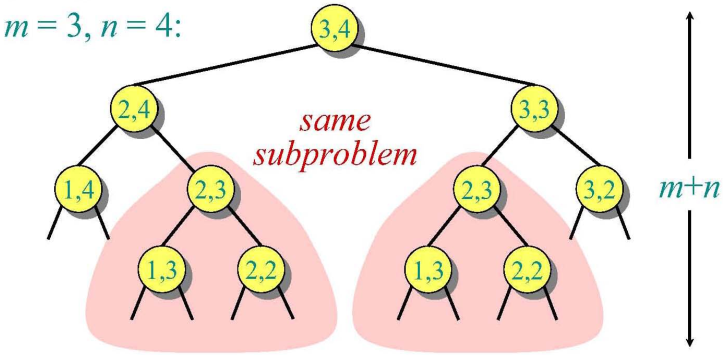


Recursion Tree



Height $= m + n \Rightarrow$ work potentially exponential.

Recursion Tree



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

Dynamic Programming Hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization

- The term **memoization** was coined by Donald Michie in 1968 and is derived from the Latin word memorandum (to be remembered), and thus carries the meaning of turning [the results of] a function into something to be remembered.
- A memoized function "remembers" the results corresponding to some set of specific inputs. Subsequent calls with remembered inputs return the remembered result rather than recalculating it, thus eliminating the primary cost of a call with given parameters from all but the first call made to the function with those parameters.

Memoization Algorithm

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
  if $c[i, j] = \text{NIL}$
    then if $x[i] = y[j]$
      then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
      else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

Dynamic Programming Algorithm

IDEA:

Compute the table bottom-up.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic Programming Algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic Programming Algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Dynamic Programming Algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

Exercise:

$O(\min\{m, n\})$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Another Example (with all the pointers)
Output: priden

j / i		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**

Other sequence questions

Edit distance: Given 2 sequences, $X = \langle x_1, \cdots, x_m \rangle$ and $Y = \langle y_1, \cdots, y_n, \rangle$, what is the minimum number of deletions, insertions, and changes that you must do to change one to another?

Protein sequence alignment: Given a score matrix on amino acid pairs, and 2 amino acid sequences, $X = \langle x_1, \cdots, x_m \rangle$ and $Y = \langle y_1, \cdots, y_n \rangle$, find the alignment with lowest scoreÉ

Conclusions on the 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).

References

- Assembly-line scheduling & Longest common subsequence
  - T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, 2001, Chapter 15, Sections 15.1 and 15.4.
- Knapsack Problem
  - U. Manber, *Introduction to Algorithms - A Creative Approach*, Addison-Wesley, 1989, Chapter 5, Section 5.10.
- Knapsack Problem & Longest common subsequence
  - M.T. Goodrich and R. Tamassia, *Algorithm Design and Applications*, Wiley, 2015, Chapter 12, Sections 12.5 and 12.6.