

Dynamic Programming

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Nov 7 & 9, 2017

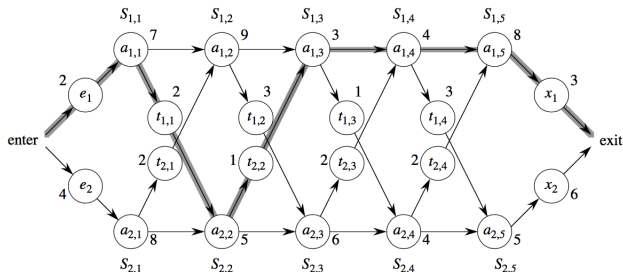
Dynamic Programming (Chap 15)

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when “programming” meant “tabular method” (like linear programming). Doesn’t really refer to computer programming.
- Used for optimization problems:
 - Find *a* solution with *the* optimal value.
 - Minimization or maximization. (We’ll see both.)

Four-step method

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.

Assembly-line scheduling



Automobile factory with two assembly lines.

- Each line has n stations: $S_{1,1}, \dots, S_{1,n}$ and $S_{2,1}, \dots, S_{2,n}$.
- 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 .
- Exit times x_1 and x_2 .
- After going through a station, can either
 - stay on same line; no cost, or
 - transfer to other line; cost after $S_{i,j}$ is $t_{i,j}$. ($j = 1, \dots, n-1$. No $t_{i,n}$, because the assembly line is done after $S_{i,n}$.)

Problem

Problem: Given all these costs (time = cost), what stations should be chosen from line 1 and from line 2 for fastest way through factory?

Try all possibilities?

- Each candidate is fully specified by which stations from line 1 are included. Looking for a subset of line 1 stations.
- Line 1 has n stations.
- 2^n subsets.
- Infeasible when n is large.

Structure of an optimal solution

Think about fastest way from entry through $S_{1,j}$.

- If $j = 1$, easy: just determine how long it takes to get through $S_{1,1}$.
- If $j \geq 2$, have two choices of how to get to $S_{1,j}$:
 - Through $S_{1,j-1}$, then directly to $S_{1,j}$.
 - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$.

Suppose fastest way is through $S_{1,j-1}$.

Key observation: We must have taken a fastest way from entry through $S_{1,j-1}$ in this solution. If there were a faster way through $S_{1,j-1}$, we would use it instead to come up with a faster way through $S_{1,j}$.

Now suppose a fastest way is through $S_{2,j-1}$. Again, we must have taken a fastest way through $S_{2,j-1}$. Otherwise use some faster way through $S_{2,j-1}$ to give a faster way through $S_{1,j}$.

Generally: An optimal solution to a problem (fastest way through $S_{1,j}$) contains within it an optimal solution to subproblems (fastest way through $S_{1,j-1}$ or $S_{2,j-1}$).

This is *optimal substructure*.

Substructure

Use optimal substructure to construct optimal solution to problem from optimal solutions to subproblems.

Fastest way through $S_{1,j}$ is either

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

Symmetrically:

Fastest way through $S_{2,j}$ is either

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

Therefore, to solve problems of finding a fastest way through $S_{1,j}$ and $S_{2,j}$, solve subproblems of finding a fastest way through $S_{1,j-1}$ and $S_{2,j-1}$.

Recursive solution

Recursive solution

Let $f_i[j]$ = fastest time to get through $S_{i,j}$, $i = 1, 2$ and $j = 1, \dots, n$.

Goal: fastest time to get all the way through = f^* .

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

$$f_1[1] = e_1 + a_{1,1}$$

$$f_2[1] = e_2 + a_{2,1}$$

For $j = 2, \dots, n$:

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

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

$f_i[j]$ gives the *value* of an optimal solution. What if we want to *construct* an optimal solution?

- $l_i[j]$ = line # (1 or 2) whose station $j-1$ is used in fastest way through $S_{i,j}$.
- In other words $S_{l_i[j],j-1}$ precedes $S_{i,j}$.
- Defined for $i = 1, 2$ and $j = 2, \dots, n$.
- l^* = line # whose station n is used.

Example

For example:

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

$$f^* = 35$$

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

$$l^* = 1$$

Go through optimal way given by l values. (Shaded path in earlier figure.)

Compute an optimal solution

Could just write a recursive algorithm based on above recurrences.

- Let $r_i(j) = \#$ of references made to $f_i[j]$.
- $r_1(n) = r_2(n) = 1$.
- $r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$ for $j = 1, \dots, n-1$.

Optimal solution

Claim

$$r_i(j) = 2^{n-j}.$$

Proof Induction on j , down from n .

Basis: $j = n$. $2^{n-j} = 2^0 = 1 = r_i(n)$.

Inductive step: Assume $r_i(j+1) = 2^{n-(j+1)}$.

$$\begin{aligned}\text{Then } r_i(j) &= r_i(j+1) + r_2(j+1) \\ &= 2^{n-(j+1)} + 2^{n-(j+1)} \\ &= 2^{n-(j+1)+1} \\ &= 2^{n-j}.\end{aligned}$$

■ (claim)

Therefore, $f_1[1]$ alone is referenced 2^{n-1} times!

So top down isn't a good way to compute $f_i[j]$.

Observation: $f_i[j]$ depends only on $f_1[j-1]$ and $f_2[j-1]$ (for $j \geq 2$).

So compute in order of *increasing* j .

Optimal solution

```
FASTEST-WAY( $a, t, e, x, n$ )  
   $f_1[1] \leftarrow e_1 + a_{1,1}$   
   $f_2[1] \leftarrow e_2 + a_{2,1}$   
  for  $j \leftarrow 2$  to  $n$   
    do if  $f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j}$   
      then  $f_1[j] \leftarrow f_1[j - 1] + a_{1,j}$   
           $l_1[j] \leftarrow 1$   
      else  $f_1[j] \leftarrow f_2[j - 1] + t_{2,j-1} + a_{1,j}$   
           $l_1[j] \leftarrow 2$   
    if  $f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j}$   
      then  $f_2[j] \leftarrow f_2[j - 1] + a_{2,j}$   
           $l_2[j] \leftarrow 2$   
      else  $f_2[j] \leftarrow f_1[j - 1] + t_{1,j-1} + a_{2,j}$   
           $l_2[j] \leftarrow 1$   
  if  $f_1[n] + x_1 \leq f_2[n] + x_2$   
    then  $f^* = f_1[n] + x_1$   
         $l^* = 1$   
    else  $f^* = f_2[n] + x_2$   
         $l^* = 2$ 
```

Go through example.

Constructing an optimal solution

PRINT-STATIONS(l, n)

$i \leftarrow l^*$

print “line ” i “, station ” n

for $j \leftarrow n$ **downto** 2

do $i \leftarrow l_i[j]$

 print “line ” i “, station ” $j - 1$

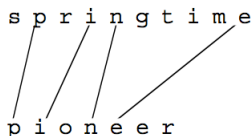
Go through example.

Time = $\Theta(n)$

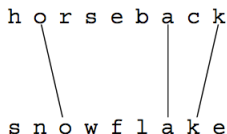
Longest common subsequence

Problem: Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

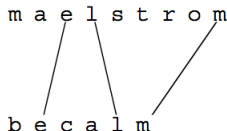
s p r i n g t i m e
p i o n e e r



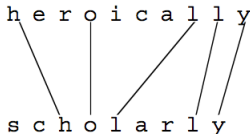
h o r s e b a c k
s n o w f l a k e



m a e l s t r o m
b e c a l m



h e r o i c a l l y
s c h o l a r l y



Longest common subsequence

Brute-force algorithm:

For every subsequence of X , check whether it's a subsequence of Y .

Time: $\Theta(n2^m)$.

- 2^m subsequences of X to check.
- Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Optimal substructure

Notation:

X_i = prefix $\langle x_1, \dots, x_i \rangle$

Y_i = prefix $\langle y_1, \dots, y_i \rangle$

Longest common subsequence

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1} .

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \dots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow$ contradicts Z being an LCS.

Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} . Clearly, it's a common subsequence. Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W . W' is common subsequence of X and Y , has length $\geq k + 1 \Rightarrow$ contradicts Z being an LCS.

2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y . Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2. ■ (theorem)

Recursive formulation

Recursive formulation

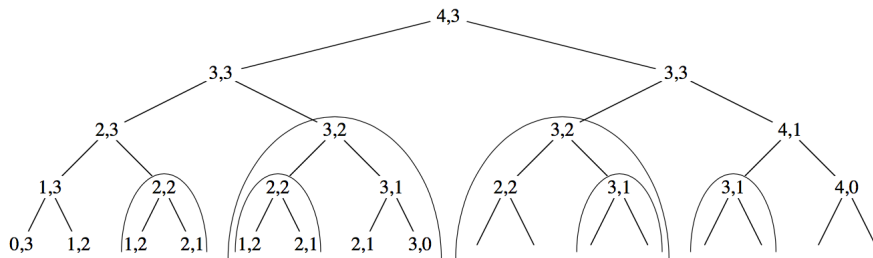
Define $c[i, j]$ = length of LCS of X_i and Y_j . We want $c[m, n]$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Again, we could write a recursive algorithm based on this formulation.

Try with bozo, bat.

Longest common subsequence



- Lots of repeated subproblems.
- Instead of recomputing, store in a table.

Longest common subsequence

Compute length of optimal solution

LCS-LENGTH(X, Y, m, n)

```
for  $i \leftarrow 1$  to  $m$ 
    do  $c[i, 0] \leftarrow 0$ 
for  $j \leftarrow 0$  to  $n$ 
    do  $c[0, j] \leftarrow 0$ 
for  $i \leftarrow 1$  to  $m$ 
    do for  $j \leftarrow 1$  to  $n$ 
        do if  $x_i = y_j$ 
            then  $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
                 $b[i, j] \leftarrow \nwarrow$ 
            else if  $c[i - 1, j] \geq c[i, j - 1]$ 
                then  $c[i, j] \leftarrow c[i - 1, j]$ 
                     $b[i, j] \leftarrow \uparrow$ 
                else  $c[i, j] \leftarrow c[i, j - 1]$ 
                     $b[i, j] \leftarrow \leftarrow$ 

return  $c$  and  $b$ 
```

Longest common subsequence

PRINT-LCS(b, X, i, j)

if $i = 0$ or $j = 0$

then return

if $b[i, j] = \nwarrow$

then PRINT-LCS($b, X, i - 1, j - 1$)

 print x_i

elseif $b[i, j] = \uparrow$

then PRINT-LCS($b, X, i - 1, j$)

else PRINT-LCS($b, X, i, j - 1$)

- Initial call is PRINT-LCS(b, X, m, n).
- $b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j .
- When $b[i, j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries with \nwarrow in them.

Demonstration: show only $c[i, j]$:

	a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2
n	0	1	1	1	1	1	2	2	2	3
k	0	1	1	1	1	1	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3
n	0	1	1	1	1	1	2	2	3	3
g	0	1	1	1	1	1	2	2	3	3

Diagram illustrating the construction of a Huffman tree from the frequency table above. The root node is labeled 'a' and has a left child 's' and a right child 'p'. Node 's' has a left child 'p' and a right child 'a'. Node 'p' has a left child 'a' and a right child 'n'. Node 'a' has a left child 'n' and a right child 'k'. Node 'n' has a left child 'i' and a right child 'n'. Node 'k' has a left child 'i' and a right child 'n'. Node 'i' has a left child 'n' and a right child 'g'. The final tree structure is shown with nodes labeled with their frequency values: 1, 2, 3, 4.

Elements of dynamic programming

Mentioned already:

- optimal substructure
- overlapping subproblems

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution. *[We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that “God” tells you what was the last choice made in an optimal solution.]*
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
 - Suppose that one of the subproblem solutions is not optimal.
 - *Cut* it out.
 - *Paste* in an optimal solution.
 - Get a better solution to the original problem. Contradicts optimality of problem solution.

Dynamic Programming

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. [*“God” is too busy to tell you what that last choice really was.*] Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?

- Keep the space as simple as possible.
- Expand it as necessary.

Examples:

Assembly-line scheduling

- Space of subproblems was fastest way from factory entry through stations $S_{1,j}$ and $S_{2,j}$.
- No need to try a more general space of subproblems.

Optimal substructure varies across problem domains:

1. *How many subproblems* are used in an optimal solution.
 2. *How many choices* in determining which subproblem(s) to use.
- Assembly-line scheduling:
 - 1 subproblem
 - 2 choices (for $S_{i,j}$ use either $S_{1,j-1}$ or $S_{2,j-1}$)
 - Longest common subsequence:
 - 1 subproblem
 - Either
 - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
 - 2 choices (if $x_i \neq y_j$, LCS of X_{i-1} and Y , and LCS of X and Y_{j-1})

Dynamic Programming

Informally, running time depends on (# of subproblems overall) \times (# of choices).

- Assembly-line scheduling: $\Theta(n)$ subproblems, 2 choices for each
 $\Rightarrow \Theta(n)$ running time.
- Longest common subsequence: $\Theta(mn)$ subproblems, ≤ 2 choices for each
 $\Rightarrow \Theta(mn)$ running time.

Dynamic programming uses optimal substructure *bottom up*.

- *First* find optimal solutions to subproblems.
- *Then* choose which to use in optimal solution to the problem.

When we look at greedy algorithms, we'll see that they work *top down*: *first* make a choice that looks best, *then* solve the resulting subproblem.

Don't be fooled into thinking optimal substructure applies to all optimization problems. It doesn't.