Dynamic Programming, Greedy Algorithm

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Nov 7 & 9, 2017

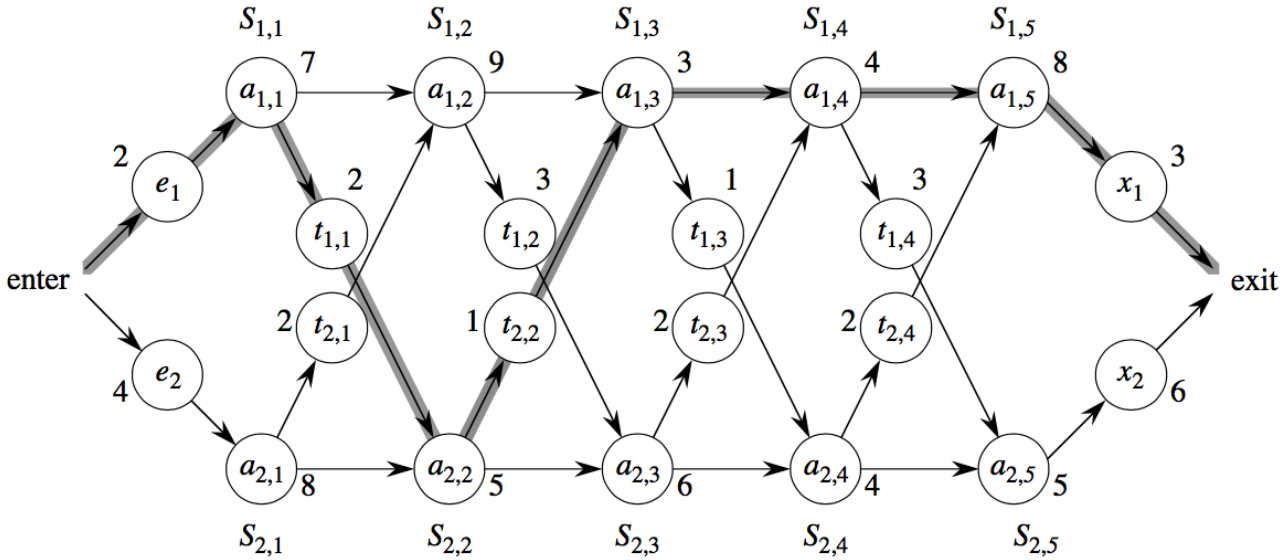
Dynamic Programming (Chap 15)

- 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: $S_{1,1}, \ldots, S_{1,n}$ 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.
- 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 $t_{i,j}$. ($j = 1, \ldots, n-1$. 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 $S_{1,j-1}$.

Substructure

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 $j = 1, \ldots, n$.

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

$$
\begin{aligned}
f^* &= \min(f_1[n] + x_1,\ f_2[n] + x_2) \\
f_1[1] &= e_1 + a_{1,1} \\
f_2[1] &= e_2 + a_{2,1}
\end{aligned}
$$

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

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

$f_i[j]$ 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 $j = 2, \ldots, n$.
- 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.)

Optimal solution

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 $j = 1, \ldots, n-1$.

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.

Optimal solution

Constructing an optimal solution

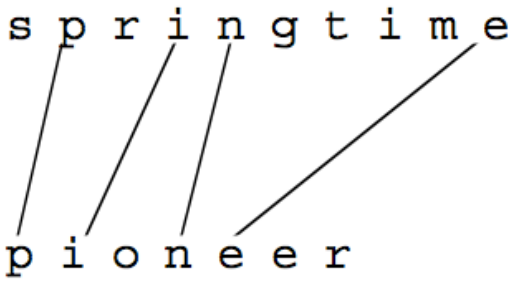
PRINT-STATIONS(l, n)
$i \leftarrow l^*$
print "line " i ", station " n
for $j \leftarrow n$ **downto** 2
    do $i \leftarrow l_i[j]$
        print "line " i ", station " $j - 1$

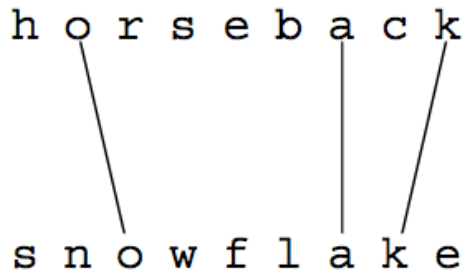
Go through example.

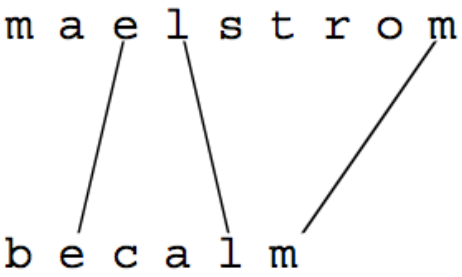
Time $= \Theta(n)$

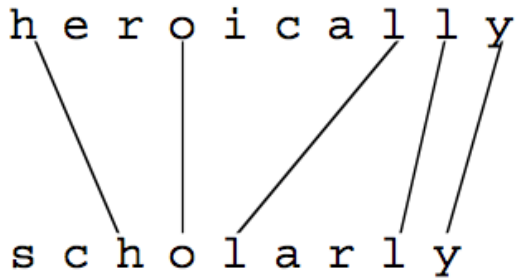
Longest common subsequence

Problem: Given 2 sequences, $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. 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.









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:

$X_i = \text{prefix } \langle x_1, \ldots, x_i \rangle$

$Y_i = \text{prefix } \langle y_1, \ldots, y_i \rangle$

Longest common subsequence

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $Z' = \langle z_1, \ldots, z_k, x_m \rangle$. 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

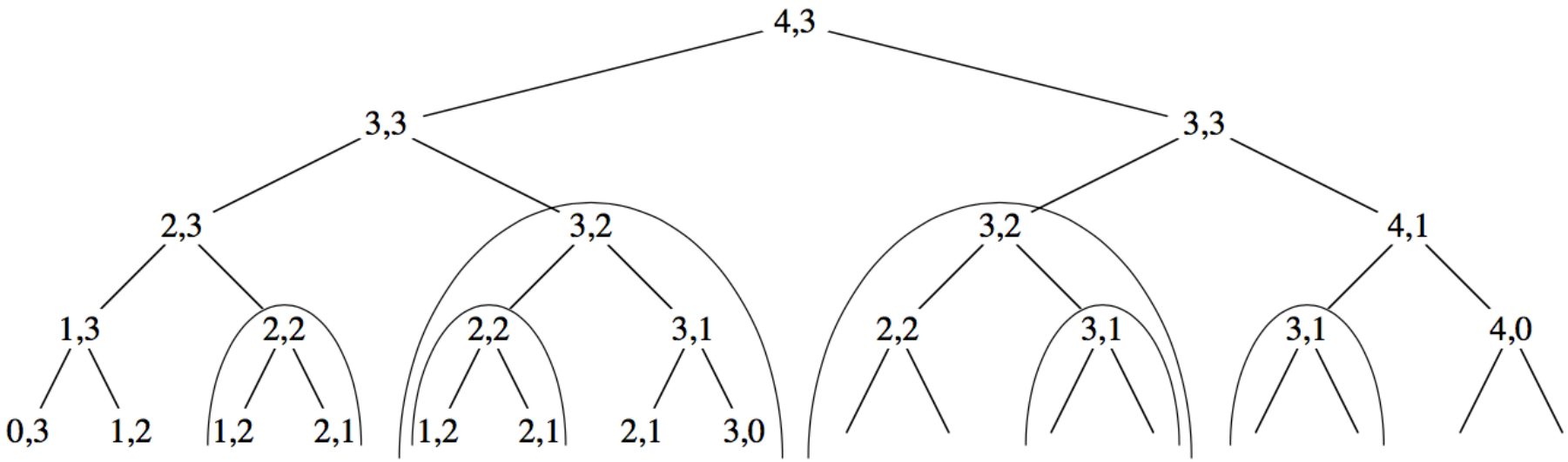
Define $c[i, j]$ = length of LCS of X_i and Y_j. We want $c[m, n]$.

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

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

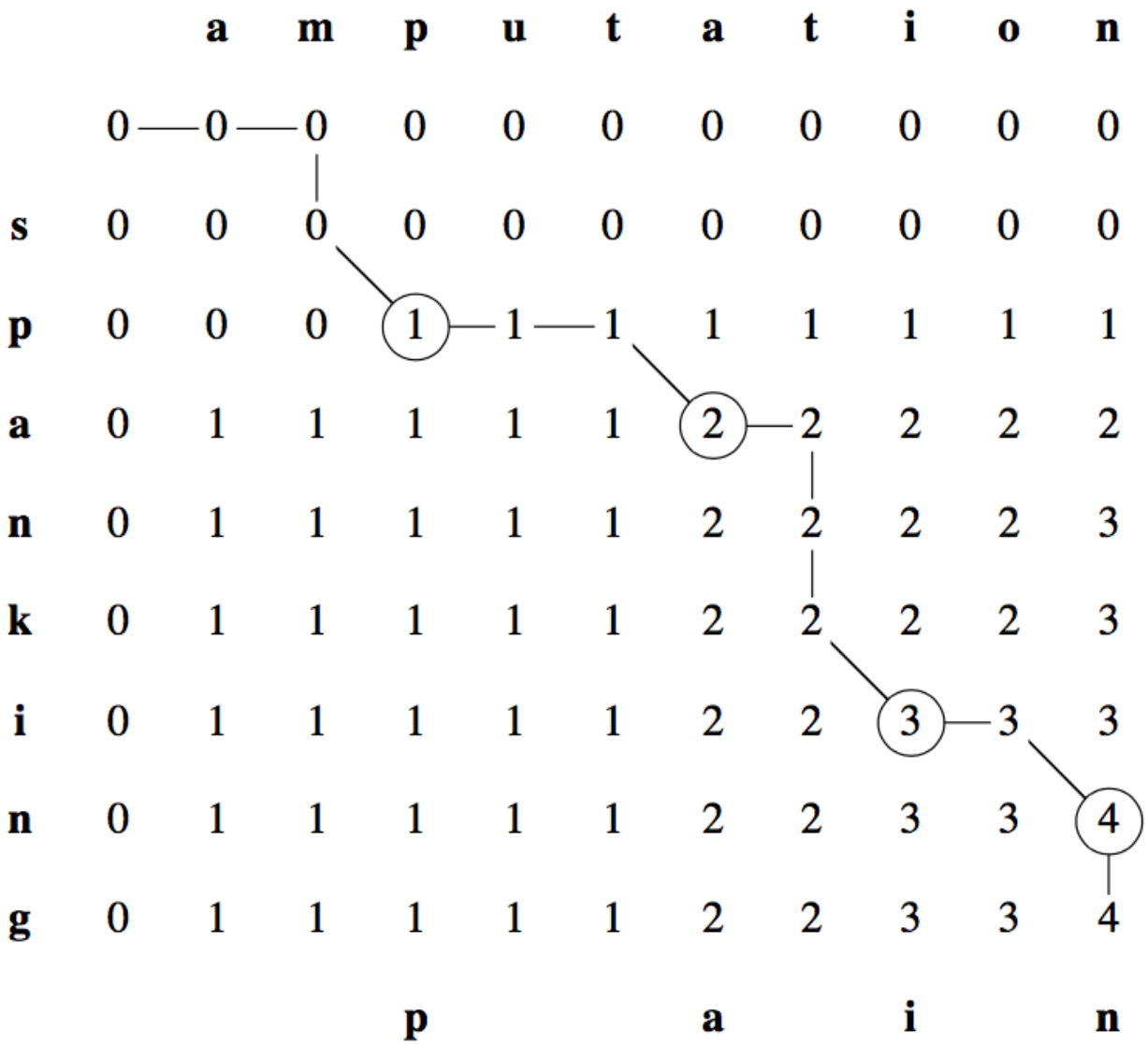
Longest common subsequence

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.

Demo

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	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4
				p			a		i		n

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.

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.

Dynamic Programming

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.

Greedy Algorithm (Chap 16)

Similar to dynamic programming.

Used for optimization problems.

Idea: When we have a choice to make, make the one that looks best *right now*. Make a *locally optimal choice* in hope of getting a *globally optimal solution*.

Greedy algorithms don't always yield an optimal solution. But sometimes they do. We'll see a problem for which they do. Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

Activity Selection

n ***activities*** require *exclusive* use of a common resource. For example, scheduling the use of a classroom.

Set of activities $S = \{a_1, \ldots, a_n\}$.

a_i needs resource during period $[s_i, f_i)$, which is a half-open interval, where s_i = start time and f_i = finish time.

Goal: Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

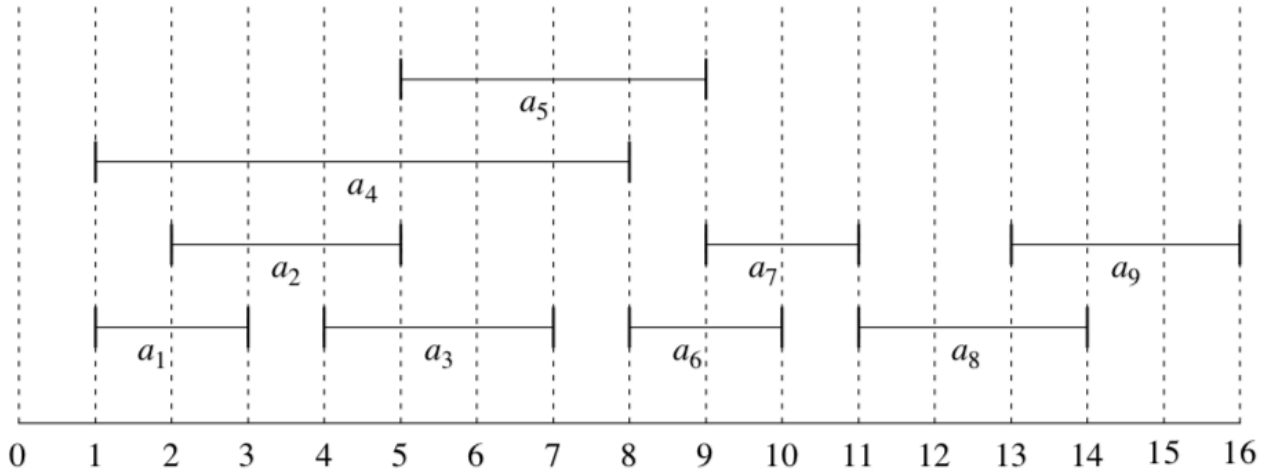
Note: Could have many other objectives:

- Schedule room for longest time.
- Maximize income rental fees.

Activity Example

Example: S sorted by finish time:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



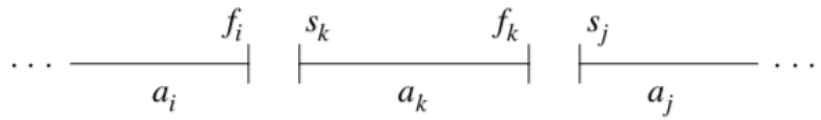
Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.

Not unique: also $\{a_2, a_5, a_7, a_9\}$.

Optimal Substructure

Optimal substructure of activity selection

$$\begin{aligned} S_{ij} &= \{a_k \in S : f_i \le s_k < f_k \le s_j\} \\ &= \text{activities that start after } a_i \text{ finishes and finish before } a_j \text{ starts .} \end{aligned}$$



Activities in S_{ij} are compatible with

- all activities that finish by f_i, and
- all activities that start no earlier than s_j.

To represent the entire problem, add fictitious activities:

$a_0 = [-\infty, 0)$

$a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$

We don't care about $-\infty$ in a_0 or “$\infty + 1$” in a_{n+1}.

Then $S = S_{0,n+1}$.

Range for S_{ij} is $0 \le i, j \le n + 1$.

Optimal Substructure

Assume that activities are sorted by monotonically increasing finish time:

$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$.

Then $i \geq j \Rightarrow S_{ij} = \emptyset$.

- If there exists $a_k \in S_{ij}$:

  $f_i \leq s_k < f_k \leq s_j < f_j \Rightarrow f_i < f_j$.

- But $i \geq j \Rightarrow f_i \geq f_j$. Contradiction.

So only need to worry about S_{ij} with $0 \leq i < j \leq n + 1$.

All other S_{ij} are $\emptyset$.

Suppose that a solution to S_{ij} includes a_k. Have 2 subproblems:

- S_{ik} (start after a_i finishes, finish before a_k starts)
- S_{kj} (start after a_k finishes, finish before a_j starts)

Optimal Substructure

Solution to S_{ij} is (solution to S_{ik}) $\cup \{a_k\} \cup$ (solution to S_{kj}).

Since a_k is in neither subproblem, and the subproblems are disjoint,

$$|\text{solution to } S| = |\text{solution to } S_{ik}| + 1 + |\text{solution to } S_{kj}| \ .$$

If an optimal solution to S_{ij} includes a_k, then the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well. Use the usual cut-and-paste argument.

Let $A_{ij} =$ optimal solution to S_{ij}.

So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ *[leave on board]* , assuming:

- S_{ij} is nonempty, and
- we know a_k.

Recursive solution

Recursive solution to activity selection

$c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_j.

- $i \geq j \Rightarrow S_{ij} = \emptyset \Rightarrow c[i, j] = 0$.

If $S_{ij} \neq \emptyset$, suppose we know that a_k is in the subset. Then

$$c[i, j] = c[i, k] + 1 + c[k, j] .$$

But of course we don't know which k to use, and so

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset . \end{cases}$$

Recursive solution

Theorem

Let $S_{ij} \neq \emptyset$, and let a_m be the activity in S_{ij} with the earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$. Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. $S_{im} = \emptyset$, so that choosing a_m leaves S_{mj} as the only nonempty subproblem.

Recursive solution

Proof

2. Suppose there is some $a_k \in S_{im}$. Then $f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$. Then $a_k \in S_{ij}$ and it has an earlier finish time than f_m, which contradicts our choice of a_m. Therefore, there is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$.
1. Let A_{ij} be a maximum-size subset of mutually compatible activities in S_{ij}.

   Order activites in A_{ij} in monotonically increasing order of finish time.

   Let a_k be the first activity in A_{ij}.

   If $a_k = a_m$, done (a_m is used in a maximum-size subset).

   Otherwise, construct $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$ (replace a_k by a_m).

Claim

Activities in A'_{ij} are disjoint.

Proof Activities in A_{ij} are disjoint, a_k is the first activity in A_{ij} to finish, $f_m \leq f_k$ (so a_m doesn't overlap anything else in A'_{ij}). ■ (claim)

Since $\left|A'_{ij}\right| = |A_{ij}|$ and A_{ij} is a maximum-size subset, so is A'_{ij}. ■ (theorem)

Recursive solution

This is great:

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j - i - 1$	1

Now we can solve *top down*:

- To solve a problem S_{ij},
  - Choose $a_m \in S_{ij}$ with earliest finish time: the ***greedy choice***.
  - Then solve S_{mj}.

What are the subproblems?

- Original problem is $S_{0,n+1}$.
- Suppose our first choice is a_{m_1}.
- Then next subproblem is $S_{m_1,n+1}$.
- Suppose next choice is a_{m_2}.
- Next subproblem is $S_{m_2,n+1}$.
- And so on.

Each subproblem is $S_{m_i,n+1}$, i.e., the last activities to finish.

And the subproblems chosen have finish times that increase.

Therefore, we can consider each activity just once, in monotonically increasing order of finish time.

Recursive solution

Easy recursive algorithm: Assumes activites already sorted by monotonically increasing finish time. (If not, then sort in $O(n \lg n)$ time.) Return an optimal solution for $S_{i,n+1}$:

REC-ACTIVITY-SELECTOR(s, f, i, n)
$m \leftarrow i + 1$
while $m \leq n$ and $s_m < f_i$ ▷ Find first activity in $S_{i,n+1}$.
  do $m \leftarrow m + 1$
if $m \leq n$
  then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return $\emptyset$

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Recursive solution

Idea: The **while** loop checks $a_{i+1}, a_{i+2}, \ldots, a_n$ until it finds an activity a_m that is compatible with a_i (need $s_m \geq f_i$).

- If the loop terminates because a_m is found ($m \leq n$), then recursively solve $S_{m,n+1}$, and return this solution, along with a_m.
- If the loop never finds a compatible a_m ($m > n$), then just return empty set.

Go through example given earlier. Should get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$—each activity examined exactly once.

Iterative solution

Can make this *iterative*. It's already almost tail recursive.

GREEDY-ACTIVITY-SELECTOR(s, f, n)

$A \leftarrow \{a_1\}$
$i \leftarrow 1$
for $m \leftarrow 2$ **to** n
    do if $s_m \geq f_i$
        then $A \leftarrow A \cup \{a_m\}$
            $i \leftarrow m$ $\quad \triangleright$ a_i is most recent addition to A
return A

Go through example given earlier. Should again get $\{a_1, a_4, a_8, a_{11}\}$.

Time: $\Theta(n)$.

Greedy strategy

The choice that seems best at the moment is the one we go with.

What did we do for activity selection?

1. Determine the optimal substructure.
2. Develop a recursive solution.
3. Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
4. Show that all but one of the subproblems resulting from the greedy choice are empty.
5. Develop a recursive greedy algorithm.
6. Convert it to an iterative algorithm.

Greedy Algorithm

At first, it looked like dynamic programming.

Typically, we streamline these steps.

Develop the substructure with an eye toward

- making the greedy choice,
- leaving just one subproblem.

For activity selection, we showed that the greedy choice implied that in S_j, only i varied, and j was fixed at $n + 1$.

We could have started out with a greedy algorithm in mind:

- Define $S_i = \{a_k \in S : f_i \leq s_k\}$.
- Then show that the greedy choice—first a_m to finish in S_i—combined with optimal solution to $S_m \Rightarrow$ optimal solution to S_i.

Greedy Algorithm

Typical streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
3. Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.

No general way to tell if a greedy algorithm is optimal, but two key ingredients are

1. greedy-choice property and
2. optimal substructure.

Greedy Algorithm

Greedy-choice property

A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Dynamic programming:

- Make a choice at each step.
- Choice depends on knowing optimal solutions to subproblems. Solve subproblems *first*.
- Solve *bottom-up*.

Greedy Algorithm

Greedy:

- Make a choice at each step.
- Make the choice *before* solving the subproblems.
- Solve *top-down*.

Typically show the greedy-choice property by what we did for activity selection:

- Look at a globally optimal solution.
- If it includes the greedy choice, done.
- Else, modify it to include the greedy choice, yielding another solution that's just as good.

Can get efficiency gains from greedy-choice property.

- Preprocess input to put it into greedy order.
- Or, if dynamic data, use a priority queue.

Optimal substructure

Just show that optimal solution to subproblem and greedy choice $\Rightarrow$ optimal solution to problem.

Greedy vs. dynamic programming

The knapsack problem is a good example of the difference.

0-1 knapsack problem:

- n items.
- Item i is worth $\$v_i$, weighs w_i pounds.
- Find a most valuable subset of items with total weight $\leq W$.
- Have to either take an item or not take it—can't take part of it.

Greedy Algorithm

Fractional knapsack problem: Like the 0-1 knapsack problem, but can take fraction of an item.

Both have optimal substructure.

But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not.

To solve the fractional problem, rank items by value/weight: v_i/w_i.

Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i.

FRACTIONAL-KNAPSACK(v, w, W)

$load \leftarrow 0$
$i \leftarrow 1$
while $load < W$ and $i \leq n$
    do if $w_i \leq W - load$
        then take all of item i
        else take $(W - load)/w_i$ of item i
        add what was taken to $load$
        $i \leftarrow i + 1$

Greedy Algorithm

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Greedy doesn't work for the 0-1 knapsack problem. Might get empty space, which lowers the average value per pound of the items taken.

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

$W = 50$.

Greedy solution:

- Take items 1 and 2.
- value $= 160$, weight $= 30$.

Have 20 pounds of capacity left over.

Optimal solution:

- Take items 2 and 3.
- value $= 220$, weight $= 50$.

No leftover capacity.

Homework 5

1. Exercise 15.1-5
2. Exercise 16.1-4

Due date: Nov 30, Thu