Dynamic Programming
Optimization Problems

• For most, the best known algorithm runs in exponential time.
• Some have quick Greedy or Dynamic Programming algorithms.
What is Dynamic Programming?

• Dynamic programming solves *optimization problems* by combining solutions to subproblems

• “Programming” refers to a tabular method with a series of choices, not “coding”
What is Dynamic Programming?

- A set of choices must be made to arrive at an optimal solution
- As choices are made, subproblems of the same form arise frequently
- The key is to *store* the solutions of subproblems to be *reused* in the future
Example 1

- Fibonacci numbers are defined by:

\[ F_0 = 0 \]
\[ F_1 = 1 \]
\[ F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2. \]
Fibonacci Example

algorithm $Fib(n)$

$\langle \text{pre-cond}\rangle$: $n$ is a positive integer.

$\langle \text{post-cond}\rangle$: The output is the $n$ Fibonacci number.

begin
    if ($n = 0$ or $n = 1$) then
        result($n$)
    else
        result($Fib(n - 1) + Fib(n - 2)$)
    end if
end algorithm
Fibonacci Example

algorithm $Fib(n)$

$\langle pre\text{-}cond \rangle$: $n$ is a positive integer.

$\langle post\text{-}cond \rangle$: The output is the $n$ Fibonacci number.

begin
    if ($n = 0$ or $n = 1$) then
        result($n$)
    else
        result($Fib(n - 1) + Fib(n - 2)$)
    end if
end algorithm

Time:

Exponential

Waste time redoing work
Memoization

**Definition:** An algorithmic technique which saves (memoizes) a computed answer for later reuse, rather than recomputing the answer.

- Memo functions were invented by Professor [Donald Michie](https://www.ed.ac.uk/) of [Edinburgh University](https://www.ed.ac.uk/).
- The idea was further developed by [Robin Popplestone](https://www.popplestone.com/), in his [Pop2](https://www.popplestone.com/) language.
- It was later integrated into LISP.
- This same principle is found at the hardware level in computer architectures which use a [cache](https://en.wikipedia.org/wiki/Cache) to store recently accessed memory locations.
Memoization in Optimization

• Remember the solutions for the subinstances
• If the same subinstance needs to be solved again, the same answer can be used.
Memoization reduces the complexity from exponential to linear!

```plaintext
algorithm Fib(n)

<pre-cond>: n is a positive integer.
(post-cond): The output is the n Fibonacci number.

begin
    (saved, fib) = Get(n)
    if( saved ) then
        result( fib )
    end if
    if( n = 0 or n = 1 ) then
        fib = n
    else
        fib = Fib(n - 1) + Fib(n - 2)
    end if
    Save(n, fib)
    result( fib )
end algorithm
```
From Memoization to Dynamic Programming

- Determine the set of subinstances that need to be solved.
- Instead of recursing from top to bottom, solve each of the required subinstances in smallest to largest order, storing results along the way.
Dynamic Programming

First determine the complete set of subinstances

\{100, 99, 98,\ldots, 0\}

Compute them in an order such that no friend must wait.

Smallest to largest
Dynamic Programming

Fill out a table containing an optimal solution for each subinstance.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>2.19x10^{20}</th>
<th>3.54x10^{20}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0,</td>
<td>1,</td>
<td>2,</td>
<td>3,</td>
<td>4,</td>
<td>5,</td>
<td>99,</td>
<td>100</td>
</tr>
</tbody>
</table>

0, 1, 2, 3, 4, 5, ..., 99, 100
Dynamic Programming

algorithm \( Fib(n) \)

\(<\text{pre-cond}>: n \text{ is a positive integer.} \)

\(<\text{post-cond}>: \text{The output is the } n \text{ Fibonacci number.} \)

begin

\text{table}[0..n] \text{fib}
\text{fib}[0] = 0
\text{fib}[1] = 1
\text{loop } i = 2..n
\text{fib}[i] = \text{fib}[i - 1] + \text{fib}[i - 2]
\text{end loop}
\text{result( fib[n] )}
end algorithm

Time Complexity?
Linear!
Dynamic Programming vs Divide-and-Conquer

• Recall the divide-and-conquer approach
  – Partition the problem into independent subproblems
  – Solve the subproblems recursively
  – Combine solutions of subproblems
  – e.g., mergesort, quicksort

• This contrasts with the dynamic programming approach
Dynamic Programming vs Divide-and-Conquer

- Dynamic programming is applicable when *subproblems are not independent*
  - i.e., subproblems share subsubproblems
  - Solve every subsubproblem only once and store the answer for use when it reappears

- A divide-and-conquer approach will do more work than necessary
A Sequence of 3 Steps

• A dynamic programming approach consists of a sequence of 3 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
Elements of Dynamic Programming

• For dynamic programming to be applicable, an optimization problem must have:

1. *Optimal substructure*  
   – An optimal solution to the problem contains within it optimal solutions to subproblems (but this may also mean a greedy strategy applies)

2. *Overlapping subproblems*  
   – The space of subproblems must be small; i.e., the same subproblems are encountered over and over
Elements of Dynamic Programming

• Dynamic programming uses optimal substructure from the bottom up:
  – *First find* optimal solutions to subproblems
  – *Then choose* which to use in optimal solution to problem.
Example 2. Making Change
Making Change

- To find the minimum number of Canadian coins to make any amount, the greedy method always works
  - At each step, just choose the largest coin that does not overshoot the desired amount

- The greedy method would not work if we did not have 5¢ coins
  - For 31 cents, the greedy method gives seven coins (25+1+1+1+1+1+1), but we can do it with four (10+10+10+1)

- The greedy method also would not work if we had a 21¢ coin
  - For 63 cents, the greedy method gives six coins (25+25+10+1+1+1), but we can do it with three (21+21+21)

- How can we find the minimum number of coins for any given set of denominations?
Example

- We assume coins in the following denominations: 1¢ 5¢ 10¢ 21¢ 25¢
- We’ll use 63¢ as our goal
A simple solution

• We always need a 1¢ coin, otherwise no solution exists for making one cent

• To make K cents:
  - If there is a K-cent coin, then that one coin is the minimum
  - Otherwise, for each value i < K,
    • Find the minimum number of coins needed to make i cents
    • Find the minimum number of coins needed to make K - i cents
  - Choose the i that minimizes this sum

• This algorithm can be viewed as divide-and-conquer, or as brute force
  – This solution is very recursive
  – It requires exponential work
  – It is infeasible to solve for 63¢
Another solution

• We can reduce the problem recursively by choosing the first coin, and solving for the amount that is left

• For 63¢:
  – One 1¢ coin plus the best solution for 62¢
  – One 5¢ coin plus the best solution for 58¢
  – One 10¢ coin plus the best solution for 53¢
  – One 21¢ coin plus the best solution for 42¢
  – One 25¢ coin plus the best solution for 38¢

• Choose the best solution from among the 5 given above

• Instead of solving 62 recursive problems, we solve 5

• This is still a very expensive algorithm
End of Lecture 18

Nov 15, 2007
CVR Vision Science Summer School

The Centre for Vision Research (CVR) at York University in Toronto offers a one-week, all-expenses-paid undergraduate summer school on the topic of vision science. This year’s program will be held June 1-6, 2008.

The program includes talks and demonstrations by CVR faculty on current research topics in vision science, as well as hands-on projects in CVR laboratories.

The curriculum reflects the wide range of active research areas at CVR, which includes basic research on vision in humans, animals, and machines, as well as applied topics such as virtual reality, visual perception in low-gravity environments and clinical vision science.

The program will accept 20 undergraduate students who are interested in pursuing a career in scientific research.

The program provides on-campus accommodations, breakfast and lunch each day, a closing banquet, and reimbursement for transportation costs.

For further information, including application instructions, see the summer school website at www.yorku.ca/cvrss.

The application deadline is February 1, 2008.
A dynamic programming solution

- Idea: Solve first for one cent, then two cents, then three cents, etc., up to the desired amount
  - *Save each answer in an array!*

- For each new amount $N$, compute all the possible pairs of previous answers which sum to $N$
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - Solution for 1¢ + solution for 12¢
      - Solution for 2¢ + solution for 11¢
      - Solution for 3¢ + solution for 10¢
      - Solution for 4¢ + solution for 9¢
      - Solution for 5¢ + solution for 8¢
      - Solution for 6¢ + solution for 7¢
An even better dynamic programming solution

- In fact, we can do a bit better than this, since the coins come in only a small number of denominations (1¢, 5¢, 10¢, 21¢, 25¢)

- For each new amount \( N \), compute cost of solution based on smaller sum + one additional coin.
  - For example, to find the solution for 13¢,
    - First, solve for all of 1¢, 2¢, 3¢, ..., 12¢
    - Next, choose the best solution among:
      - solution for 12¢ + 1¢ coin
      - solution for 8¢ + 5¢ coin
      - solution for 3¢ + 10¢ coin
Making Change: Recurrence Relation

Let \( sum \) = value of change to return

Let \( d[1...n] \) = denominations available

Let \( \text{mincoins}(sum) \) = minimum number of coins required to make change totalling sum.

Let \( \text{onecoin}(sum) \) = one coin in optimal set of coins to make change totalling sum.

Then

\[
\text{mincoins}(sum) = \min_{d \leq \text{sum}} \left( \text{mincoins}(sum - d) + 1 \right)
\]

\[
\text{onecoin}(sum) = \arg\min_{d \leq \text{sum}} \left( \text{mincoins}(sum - d) \right)
\]
function coins = makechange(d, sum)
%precondition: d=set of denominations (must include penny), sum=change to be made
%postcondition: coins = a minimal set of coins summing to sum
mincoins(0) = 0
mincoins(1…sum) = ∞
for i = 1:sum
    %LI: mincoins(0…i-1) holds the min number of coins required to make change of (0…i-1).
    %     onecoin(1…i-1) holds the value of one coin in a minimal set of coins making the correct change.
    for j = 1:length(d) %try each denomination
        if d(j) <= i & mincoins(i-d(j)) + 1 < mincoins(i)
            mincoins(i) = mincoins(i-d(j)) + 1 %best solution so far
            onecoin(i) = d(j)
        end
    end
end
ncoins = mincoins(sum)
change = sum
for i = 1:ncoins %recover coins in optimal set
    coins(i) = onecoin(change)
    change = change - coins(i)
How good is the algorithm?

• The first algorithm is exponential, with a base proportional to sum (e.g., 63).

• The second algorithm is much better – exponential with a base proportional to the number of denominations (e.g., 5).

• The dynamic programming algorithm is

\[ O(\text{sum} \times \text{number of denominations}) \]
Elements of Dynamic Programming

• For dynamic programming to be applicable, an optimization problem must have:

1. *Optimal substructure*
   
   – An optimal solution to the problem contains within it optimal solutions to subproblems (but this may also mean a greedy strategy applies)
Elements of Dynamic Programming

• Dynamic programming uses optimal substructure from the bottom up:
  – *First* find optimal solutions to subproblems
  – *Then choose* which to use in optimal solution to problem.
Example Proof of Optimal Substructure

- Consider the problem of making \( Nc \) with the fewest number of coins
  - Either there is an \( Nc \) coin, or
  - The set of coins \( n \) making up an optimal solution for \( Nc \) can be divided into two nonempty subsets, \( n_1 \) and \( n_2 \), which make \( N_1c \) and \( N_2c \) change respectively, where \( N_1c + N_2c = Nc \).
  - If either \( N_1c \) or \( N_2c \) can be made with fewer coins, then clearly \( Nc \) can be made with fewer coins, hence solution was *not* optimal.
  - Thus each subset \( n_1 \) and \( n_2 \) must themselves be optimal solutions to the subproblems of making \( N_1c \) and \( N_2c \) change, respectively.
Optimal Substructure

• Optimal substructure means that
  – Every optimal solution to a problem contains...
  – ...optimal solutions to subproblems

• Optimal substructure does not mean that
  – If you have optimal solutions to all subproblems...
  – ...then you can combine any of them to get an optimal solution to a larger problem.

• Example: In Canadian coinage,
  – The optimal solution to 7¢ is 5¢ + 1¢ + 1¢, and
  – The optimal solution to 6¢ is 5¢ + 1¢, but
  – The optimal solution to 13¢ is not 5¢ + 1¢ + 1¢ + 5¢ + 1¢

• But there is some way of dividing up 13¢ into subsets with optimal solutions (say, 11¢ + 2¢) that will give an optimal solution for 13¢
  – Hence, the making change problem exhibits optimal substructure.
Optimal Substructure

• Thus the step of choosing which subsolutions to combine is a key part of a dynamic programming algorithm.
Don’t all problems have this optimal substructure property?
Longest simple path

• Consider the following graph:

• The longest simple path (path not containing a cycle) from A to D is A B C D

• However, the subpath A B is not the longest simple path from A to B (A C B is longer)

• The principle of optimality is not satisfied for this problem

• Hence, the longest simple path problem cannot be solved by a dynamic programming approach
Example 2. Knapsack Problem

Get as much value as you can into the knapsack
The (General) 0-1 Knapsack Problem

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 $\leq W$.
• $v_i$, $w_i$ and $W$ are all integers.
• Have to either take an item or not take it - can’t take part of it.

Is there a greedy solution to this problem?
What are good greedy local choices?

• Select most valuable object?
• Select smallest object?
• Select object most valuable by weight?
Some example problem instances

Let $W =$ Capacity of knapsack $= 10$kg

Problem Instance 1:  
$v_1 = $60, $w_1 = 6$kg  
$v_2 = $50, $w_2 = 5$kg  
$v_3 = $50, $w_3 = 5$kg

Problem Instance 2:  
$v_1 = $60, $w_1 = 10$kg  
$v_2 = $50, $w_2 = 9$kg

Problem Instance 3:  
$v_1 = $60, $w_1 = 6$kg  
$v_2 = $40, $w_2 = 5$kg  
$v_3 = $40, $w_3 = 5$kg

• Select most valuable object?  
• Select smallest object?  
• Select object most valuable by weight?

All Fail!
Simplified 0-1 Knapsack Problem

- The general 0-1 knapsack problem cannot be solved by a greedy algorithm.
- What if we make the problem simpler:

Suppose \( v_i = w_i \)

- Can this simplified knapsack problem be solved by a greedy algorithm?
- No!
Some example problem instances

Let $W$ = Capacity of knapsack = 10kg

Problem Instance 1:
$v_1 = w_1 = 6$
$v_2 = w_2 = 5$
$v_3 = w_3 = 5$

Problem Instance 2:
$v_1 = w_1 = 10$
$v_2 = w_2 = 9$

- Select largest (most valuable) object?
- Select smallest object?

\{ Both Fail! \}
Approximate Greedy Solution

• For the simplified knapsack problem:
  • the greedy solution (taking the most valuable object first) isn’t that bad:

\[ \hat{V} \geq \frac{1}{2} V, \text{ where} \]

\[ \hat{V} = \text{Total value of items selected by greedy algorithm} \]
\[ V = \text{Total value of items selected by optimal algorithm} \]
End of Lecture 19

Nov 20, 2007
YORK UNIVERSITY
Office of the Dean
FACULTY OF SCIENCE AND ENGINEERING
MEMORANDUM

TO: Distribution
FROM: Walter P. Tholen, Associate Dean (Research & Faculty Affairs)
DATE: November 15, 2007
SUBJECT: NSERC Undergraduate Student Research Awards (NSERC USRA)

It is a pleasure to bring to your attention that for 2008-2009 York has been allocated 51 NSERC USRAs. Considering not all offers will be accepted, we should try to recruit at least 70 qualified applicants. Please bring these awards to the attention of qualified students as soon as possible. Further information on NSERC USRAs can be found on the NSERC website. The application form and instructions (Form 202) are now available only on the NSERC website and can be printed from there. Please read all instructions before completing the application form.

I. NSERC USRA in Universities

(a) The value of the award from NSERC is $4,500 for 16 consecutive weeks. Faculty members must pay the student at least 25% ($1,125) on top of this. If a student is selected from another university to hold their award at York, the supervisor will be responsible for paying at least $787.50 extra; i.e., 4% vacation pay and 10% for benefits. Travel allowance may also be available when applicable.

(b) At York, the award can only be held during the summer session.

(c) NSERC expects students to work the full term. NSERC may approve shorter tenure in exceptional circumstances (these circumstances must be approved by NSERC), so the departments must make the students aware that the awards are for the full 16 weeks.

(d) Students must return the completed applications to their departmental offices by January 25, 2008. Transcripts for York students can be printed by the departments as NSERC does not require that an official transcript be sent to them. Departments must submit their completed applications and transcripts, along with their rankings, to my office no later than February 8, 2008.
Approximate Greedy Solution

Claim: \( \hat{V} \geq \frac{1}{2} V \)

Proof:

Let \( W \) = capacity of knapsack.
Let \( s \) = value (weight) of object in optimal solution but not selected by greedy algorithm.

Suppose \( \hat{V} < \frac{1}{2} V \)

Then \( s < \frac{1}{2} V \) (since object was not selected by greedy algorithm)

But since \( \hat{V} + s < V \leq W \), object would be selected by greedy algorithm.

→ Contradiction!

And running time \( \in O(n \log n) \)
where \( n \) = number of items
Dynamic Programming Solution

- The General 0-1 Knapsack Problem can be solved by dynamic programming.

Let $W =$ capacity of knapsack (kg)
Let $(v_i, w_i) =$ value ($) and weight (kg) of item $i \in [1...n]$
Let $c[i, w] =$ value of optimal solution for knapsack of capacity $w$ and items drawn from $[1...i]$

Then $c[i, w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
\max (v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w_i \leq w \\
c[i-1, w] & \text{if } w_i > w
\end{cases}$
Correctness

Let $W$ = capacity of knapsack (kg)
Let $(v_i, w_i) =$ value ($) and weight (kg) of item $i \in [1...n]$
Let $c[i, w] =$ value of optimal solution for knapsack of capacity $w$ and items drawn from $[1...i]$

Then $c[i, w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
\max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i \\
\end{cases}$

Idea: $c[i-1, w] =$ value of optimal solution for capacity $w$ and items drawn only from $[1...i-1]$

What happens when we are also allowed to consider item $i$?

Case 1. Optimal solution does not include item $i$.
   Total value is the same as before.

Case 2. Optimal solution does include item $i$.
   Total value is:
   Value of item $i$
   + Value of optimal solution for remaining capacity of knapsack and allowable items

One of these must be true!
Bottom-Up Computation

Let $W =$ capacity of knapsack (kg)
Let $(v_i, w_i) =$ value ($) and weight (kg) of item $i \in [1...n]$
Let $c[i, w] =$ value of optimal solution for knapsack of capacity $w$ and items drawn from $[1...i]$

Then $c[i, w] =$

\[
\begin{cases}
0 & \text{if } i = 0 \text{ or } w = 0 \\
c[i-1, w] & \text{if } w_i > w \\
\max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geq w_i
\end{cases}
\]

Need only ensure that $c[i-1, v]$ has been computed $\forall v \leq w$

<table>
<thead>
<tr>
<th>$c[i, w]$</th>
<th>$w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>Allowed Items</td>
</tr>
<tr>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
</tr>
<tr>
<td>2</td>
<td>{1 2}</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>$i$</td>
<td>{1 2 … $i$}</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>$n$</td>
<td>{1 2 … $n$}</td>
</tr>
</tbody>
</table>
**Integer Knapsack Solution**

\[
\text{DYNAMIC-0-1-KNAPSACK}(v, w, n, W)
\]

\[
\begin{align*}
\text{for } w & \leftarrow 0 \text{ to } W \\
& \quad \text{do } c[0, w] \leftarrow 0 \\
\text{for } i & \leftarrow 1 \text{ to } n \\
& \quad \text{do } c[i, 0] \leftarrow 0 \\
& \quad \text{for } w \leftarrow 1 \text{ to } W \\
& \quad \quad \text{do if } w_i \leq w \\
& \quad \quad \quad \text{then if } v_i + c[i - 1, w - w_i] > c[i - 1, w] \\
& \quad \quad \quad \quad \text{then } c[i, w] \leftarrow v_i + c[i - 1, w - w_i] \\
& \quad \quad \quad \quad \text{else } c[i, w] \leftarrow c[i - 1, w] \\
& \quad \quad \quad \text{else } c[i, w] \leftarrow c[i - 1, w]
\end{align*}
\]

Value=0 if no items or no knapsack.
**Integer Knapsack Solution**

**Dynamic-0-1-Knapsack** \((v, w, n, W)\)

\[
\text{for } w \leftarrow 0 \text{ to } W \\
\quad \text{do } c[0, w] \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
\quad \text{do } c[i, 0] \leftarrow 0 \\
\quad \text{for } w \leftarrow 1 \text{ to } W \\
\quad \quad \text{do if } w_i \leq w \\
\quad \quad \quad \text{then if } v_i + c[i - 1, w - w_i] > c[i - 1, w] \\
\quad \quad \quad \quad \text{then } c[i, w] \leftarrow v_i + c[i - 1, w - w_i] \\
\quad \quad \quad \quad \text{else } c[i, w] \leftarrow c[i - 1, w] \\
\quad \quad \quad \text{else } c[i, w] \leftarrow c[i - 1, w] \\
\]

Recurrence relation

Fill in table row-wise
Then \( c[i,w] = \)

\[
\begin{align*}
0 & \quad \text{if } i = 0 \text{ or } w = 0 \\
c[i-1,w] & \quad \text{if } w_i > w \\
\max(v_i + c[i-1,w-w_i],c[i-1,w]) & \quad \text{if } i > 0 \text{ and } w \geq w_i
\end{align*}
\]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(v)</th>
<th>(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c[i,w])</th>
<th>Allowed Items</th>
<th>(w)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>(0)</td>
<td>(1)</td>
</tr>
<tr>
<td>0</td>
<td>{}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>{1 2}</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>{1 2 3}</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>{1 2 3 4}</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>{1 2 3 4 5}</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>{1 2 3 4 5 6}</td>
<td>0</td>
</tr>
</tbody>
</table>
Solving for the Items to Pack

Then \( c[i,w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
\min \{ c[i-1,w], \max(v_i + c[i-1, w-w_i], c[i-1,w]) \} & \text{if } i > 0 \text{ and } w \geq w_i 
\end{cases} \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( v )</th>
<th>( w )</th>
<th>( c[i,w] )</th>
<th>( w )</th>
<th>( i )</th>
<th>( i )</th>
<th>( i )</th>
<th>( i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{1 2}</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>{1 2 3}</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>{1 2 3 4}</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>{1 2 3 4 5}</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>{1 2 3 4 5 6}</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>11</td>
<td>15</td>
</tr>
</tbody>
</table>

\( i = n \)

\( w = W \)

items = \{\}\n
loop for \( i = n \) downto 1

if \( c[i,w] > c[i-1,w] \)

items = items + \{i\}

\( w = w - w_i \)
Second Example

**Allowed Items**

Then \( c[i,w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
\max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } w_i > w \\
c[i-1, w] & \text{if } i > 0 \text{ and } w \geq w_i 
\end{cases} \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( v )</th>
<th>( w )</th>
<th>( c[i,w] )</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>0 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2 3</td>
<td>0 0 1 4 4 5 5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3 4 5</td>
<td>0 2 2 4 6 6 7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>4 5 6</td>
<td>0 2 2 4 6 7 7</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>5 6 7</td>
<td>0 2 2 4 6 7 8</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>6 7 8</td>
<td>0 2 2 4 6 7 8</td>
</tr>
</tbody>
</table>
Knapsack Problem: Running Time

- Running time $\Theta(n \times W)$. (cf. Making change $\Theta(d \times \text{sum})$).
  - Not polynomial in input size!
End of Lecture 20

Nov 22, 2007
Recall: Knapsack Problem

Then \( c[i,w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
\max(v_i + c[i-1, w-w_i], c[i-1, w]) & \text{if } w_i > w \\
c[i-1, w] & \text{if } i > 0 \text{ and } w \geq w_i 
\end{cases} \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( v )</th>
<th>( w )</th>
<th>( c[i,w] )</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{}</td>
<td>0</td>
<td>0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{1}</td>
<td>0</td>
<td>0 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{1 2}</td>
<td>0</td>
<td>0 1 4 4 5 5 5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{1 2 3}</td>
<td>0</td>
<td>2 2 4 6 6 7 7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>{1 2 3 4}</td>
<td>0</td>
<td>2 2 4 6 7 7 7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>{1 2 3 4 5}</td>
<td>0</td>
<td>2 2 4 6 7 8 8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>{1 2 3 4 5 6}</td>
<td>0</td>
<td>2 2 4 6 7 8 8</td>
<td></td>
</tr>
</tbody>
</table>

Capacity \( W = 6 \)
Observation from Last Day (Jonathon):
We could still implement this recurrence relation directly as a recursive program.

Then \( c[i,w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0 \\
c[i-1,w] & \text{if } w_i > w \\
\max(v_i + c[i-1,w-w_i],c[i-1,w]) & \text{if } i > 0 \text{ and } w \geq w_i 
\end{cases} \)

<table>
<thead>
<tr>
<th>( c[i,w] )</th>
<th>( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>Allowed Items</td>
</tr>
<tr>
<td>0 {}</td>
<td></td>
</tr>
<tr>
<td>1 {1}</td>
<td></td>
</tr>
<tr>
<td>2 {1 2}</td>
<td></td>
</tr>
<tr>
<td>3 {1 2 3}</td>
<td></td>
</tr>
<tr>
<td>4 {1 2 3 4}</td>
<td></td>
</tr>
<tr>
<td>5 {1 2 3 4 5}</td>
<td></td>
</tr>
<tr>
<td>6 {1 2 3 4 5 6}</td>
<td></td>
</tr>
</tbody>
</table>
Recall: Memoization in Optimization

• Remember the solutions for the subinstances

• If the same subinstance needs to be solved again, the same answer can be used.
Memoization

Algorithm $Fib(n)$

$\langle pre-cond \rangle$: $n$ is a positive integer.

$\langle post-cond \rangle$: The output is the $n$ Fibonacci number.

begin
  $\langle saved, fib \rangle = Get(n)$
  if( $\langle saved \rangle$ then
    result( $\langle fib \rangle$)
  end if
  if( $n = 0$ or $n = 1$ ) then
    $fib = n$
  else
    $fib = Fib(n - 1) + Fib(n - 2)$
  end if
  Save($n$, $fib$)
  result( $\langle fib \rangle$)
end algorithm

Memoization reduces the complexity from exponential to linear!
From Memoization to Dynamic Programming

• Determine the set of subinstances that need to be solved.

• Instead of recursing from top to bottom, solve each of the required subinstances in smallest to largest order, storing results along the way.
Dynamic Programming Examples

1. Fibonacci numbers
2. Making change
3. 0-1 Knapsack problem
4. Activity Scheduling with profits
Recall: The Activity (Job/Event) Selection Problem

Ingredients:

- **Instances**: Events with starting and finishing times $\langle<s_1,f_1>,<s_2,f_2>,\ldots,<s_n,f_n>\rangle$.
- **Solutions**: A set of events that do not overlap.
- **Value of Solution**: The number of events scheduled.
- **Goal**: Given a set of events, schedule as many as possible.
From Previous Lecture:
Problem can be solved by greedy algorithm

Greedy Criteria: **Earliest Finishing Time**

Motivation: Schedule the event that will free up your room for someone else as soon as possible. **Works!**
But what if activities have different values?

Activity Selection with Profits:

**Input:** information \((s_i, f_i, g_i)\) about \(n\) activities, where
- \(s_i = \) start time of activity \(i\)
- \(f_i = \) finishing time of activity \(i\)
- \(g_i = \) value (profit) of activity \(i\)

A **feasible schedule** is a set \(S \subseteq \{1, 2, \ldots, n\}\) such that \(\forall i, j \in S,\) activities \(i\) and \(j\) do not conflict.

**Output:** A feasible schedule \(S\) with maximum profit

\[
P(S) = \sum_{i \in S} g_i
\]
Will a greedy algorithm based on finishing time still work?

\[ g_1 = 1 \quad g_2 = 10 \quad g_3 = 1 \]

No!
Dynamic Programming Solution

Precomputation:

1. Sort activities according to finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$ \hspace{1cm} $O(n \log n)$

2. $\forall i \in \{1, \ldots, n\}$, compute $H(i) = \max \{l \in \{1, 2, \ldots, i-1\} \mid f_l \leq s_i\}$ \hspace{1cm} $O(n \log n)$

i.e. $H(i)$ is the last event that ends before event $i$ starts.

Time?
Step 1. Define an array of values to compute

\[ \forall i \in \{0,\ldots,n\}, \ A(i) = \text{largest profit attainable from the (feasible) scheduling of a subset of activities from } \{1, 2, \ldots, i\} \]

Ultimately, we are interested in \( A(n) \)
Step 2. Provide a Recurrent Solution

1. Sort activities according to finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \) \( (O(n \log n)) \)

2. \( \forall i \in \{1, \ldots, n\} \), compute \( H(i) = \max \{l \in \{1, 2, \ldots, i - 1\} \mid f_l \leq s_i\} \) \( (O(n \log n)) \)
   
i.e. \( H(i) \) is the last event that ends before event \( i \) starts.

\[
A(0) = 0 \\
A(i) = \max \{A(i - 1), g_i + A(H(i))\}, \forall i \in \{1, \ldots, n\}
\]

One of these must be true!

Decide not to schedule activity \( i \)

Profit from scheduling activity \( i \)

Optimal profit from scheduling activities that end before activity \( i \) begins
Step 3. Provide an Algorithm

function A=actselwithp(g, H, n)
% assumes inputs sorted by finishing time
A(0)=0
for i=1:n
    A(i)=max(A(i-1), g(i)+A(H(i)))
end

Running time?  $O(n)$
Step 4. Compute Optimal Solution

Invoke with: `printasp(A,H,n,'Activities to Schedule:')`

```matlab
function actstring = printasp(A,H,i,actstring)

if i == 0
    return
else

    if A(i) > A(i-1)
        actstring = printasp(A,H,H(i),actstring)
        actstring = [actstring, sprintf('%d ', i)]
    else
        actstring = printasp(A,H,i-1,actstring)
    end

end

Running time? \( O(n) \)
```
### Example

<table>
<thead>
<tr>
<th>Activity $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start $s_i$</strong></td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>Finish $f_i$</strong></td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td><strong>Profit $g_i$</strong></td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td><strong>$H(i)$</strong></td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Activity $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start $s_i$</strong></td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td><strong>Finish $f_i$</strong></td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td><strong>Profit $g_i$</strong></td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td><strong>$H(i)$</strong></td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Activity $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start $s_i$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Finish $f_i$</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Profit $g_i$</td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>$H(i)$</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Activity $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start $s_i$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Finish $f_i$</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Profit $g_i$</td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>$H(i)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Activity $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start $s_i$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Finish $f_i$</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>Profit $g_i$</td>
<td>20</td>
<td>30</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>$H(i)$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$A(0) = 0$

$A(1) = \max \{0, 20 + A(H(1))\} = 20$

$A(2) = \max \{20, 30 + A(H(2))\} = 30$

$A(3) = \max \{30, 20 + A(H(3))\} = 40$

$A(4) = \max \{40, 30 + A(H(4))\} = 40$
Dynamic Programming Examples

1. Fibonacci numbers
2. Making change
3. 0-1 Knapsack problem
4. Activity scheduling with profits
5. Longest common subsequence
Longest Common Subsequence

- **Input:** 2 sequences, \( X = x_1, \ldots, x_m \) and \( Y = y_1, \ldots, y_n \).

- **Output:** a subsequence common to both whose length is longest.

- **Note:** A subsequence doesn’t have to be consecutive, but it has to be in order.
Example 3. Longest Common Subsequence
Brute-force Algorithm

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

Time: $(n2^m)$.

$2^m$ 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.
Step 1. Define Data Structure

• Input: 2 sequences, \( X = x_1, \ldots, x_m \) and \( Y = y_1, \ldots, y_n \).

Notation:
\[
X_i = \text{prefix } \langle x_1, \ldots, x_i \rangle \\
Y_i = \text{prefix } \langle y_1, \ldots, y_i \rangle
\]

Let \( c(i, j) = \text{length of LCS for } X_i \text{ and } Y_j \)

Ultimately, we are interested in \( c(m, n) \).
Step 2. Define Recurrence

Case 1. Input sequence is empty

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

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

Case 2.
Last elements match:
must be part of an LCS

\[\text{X} \quad \text{maelstrom} \quad \text{Y} \]

\[X_{i-1} \quad Y_{j-1} \]

\[\text{becalm} \quad \text{Z} \quad \_\_\_\_\_\_\_\_\_\_\_\_\_m\]
Recurrence

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

Case 3. Last elements don't match: at most one of them is part of LCS
Step 3. Provide an Algorithm

LCS-LENGTH(X, Y)

1  \( m \leftarrow \text{length}[X] \)
2  \( n \leftarrow \text{length}[Y] \)
3  \textbf{for} i \leftarrow 1 \textbf{to} m
4      \textbf{do} c[i, 0] \leftarrow 0
5  \textbf{for} j \leftarrow 0 \textbf{to} n
6      \textbf{do} c[0, j] \leftarrow 0
7  \textbf{for} i \leftarrow 1 \textbf{to} m
8      \textbf{do for} j \leftarrow 1 \textbf{to} n
9          \textbf{if} x_i = y_j
10             \textbf{then} c[i, j] \leftarrow c[i - 1, j - 1] + 1
11                b[i, j] \leftarrow “\text\{\textbackslash\}\text\{\textup\}”
12          \textbf{else if} c[i - 1, j] \geq c[i, j - 1]
13             \textbf{then} c[i, j] \leftarrow c[i - 1, j]
14                b[i, j] \leftarrow “↑”
15          \textbf{else} c[i, j] \leftarrow c[i, j - 1]
16                b[i, j] \leftarrow “←”
17  \textbf{return} c \text{ and } b

Running time? \( O(mn) \)
Step 4. Compute Optimal Solution

PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = “\”
4     then PRINT-LCS(b, X, i - 1, j - 1)
5        print x_i
6  elseif b[i, j] = “↑”
7     then PRINT-LCS(b, X, i - 1, j)
8  else PRINT-LCS(b, X, i, j - 1)

• Initial call is PRINT-LCS(b, X, m, n).

Running time? \( O(m+n) \)
Example

- $b[i, j]$ points to table entry whose subproblem we used in solving LCS of $X_i$ and $Y_j$.

- When $b[i, j] = \swarrow$, we have extended LCS by one character. So longest common subsequence = entries with $\swarrow$ in them.