Dynamic Programming

Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of \textcolor{red}{A}, \textcolor{blue}{C}, \textcolor{green}{G}, and \textcolor{purple}{T} characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.
Application: DNA Sequence Alignment

Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other. A brute-force search would take exponential time (in fact $O(n2^n)$), but we can do much better using dynamic programming.

Figure 12.1: Two DNA sequences, X and Y, and their alignment in terms of a longest subsequence, GTCGTCCGAAGCGCGCCGCAA, that is common to these two strings.
Warm-up: Matrix Chain-Products

- **Dynamic Programming** is a general algorithm design paradigm.
  - Rather than give the general structure, let us first give a motivating example:
  - **Matrix Chain-Products**

- **Review: Matrix Multiplication.**
  - $C = A \times B$
  - $A$ is $d \times e$ and $B$ is $e \times f$

\[
C[i, j] = \sum_{k=0}^{e-1} A[i, k] \times B[k, j]
\]

- $O(denf)$ time
Matrix Chain-Products

Matrix Chain-Product:
- Compute $A = A_0 * A_1 * ... * A_{n-1}$
- $A_i$ is $d_i \times d_{i+1}$
- Problem: How to parenthesize?

Example
- $B$ is $3 \times 100$
- $C$ is $100 \times 5$
- $D$ is $5 \times 5$
- $(B*C)*D$ takes $1500 + 75 = 1575$ ops
- $B*(C*D)$ takes $1500 + 2500 = 4000$ ops
An Enumeration Approach

Matrix Chain-Product Alg.:
- Try all possible ways to parenthesize $A = A_0 \cdot A_1 \cdots \cdot A_{n-1}$
- Calculate number of ops for each one
- Pick the one that is best

Running time:
- The number of paranethesizations is equal to the number of binary trees with $n$ nodes (why?)
- This is exponential!
- It is called the Catalan number, and it is almost $4^n$.
- This is a terrible algorithm!
A Greedy Approach

Idea #1: repeatedly select the product that uses (up) the most operations.

Counter-example:
- A is $10 \times 5$
- B is $5 \times 10$
- C is $10 \times 5$
- D is $5 \times 10$

Greedy idea #1 gives $(A*B)*(C*D)$, which takes $500+1000+500 = 2000$ ops
- $A*((B*C)*D)$ takes $500+250+250 = 1000$ ops
Another Greedy Approach

- Idea #2: repeatedly select the product that uses the fewest operations.

- Counter-example:
  - A is 101 × 11
  - B is 11 × 9
  - C is 9 × 100
  - D is 100 × 99
  - Greedy idea #2 gives A*((B*C)*D)), which takes 109989+9900+108900=228789 ops
  - (A*B)*(C*D) takes 9999+89991+89100=189090 ops
  - The greedy approach is not giving us the optimal value.
A “Recursive” Approach

Define subproblems:
- Find the best parenthesization of $A_i*A_{i+1}*...*A_j$.
- Let $N_{i,j}$ denote the number of operations done by this subproblem.
- The optimal solution for the whole problem is $N_{0,n-1}$.

Subproblem optimality: The optimal solution can be defined in terms of optimal subproblems
- There has to be a final multiplication (root of the expression tree) for the optimal solution.
- Say, the final multiply is at index $i$: $(A_0*...*A_i)(A_{i+1}*...*A_{n-1})$.
- Then the optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiply.
- If the global optimum did not have these optimal subproblems, we could define an even better “optimal” solution.
The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiply is at.

Let us consider all possible places for that final multiply:

- Recall that $A_i$ is a $d_i \times d_{i+1}$ dimensional matrix.
- So, a characterizing equation for $N_{i,j}$ is the following:

$$N_{i,j} = \min_{i \leq k < j} \left\{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \right\}$$

Note that subproblems are not independent--the subproblems overlap.
Since subproblems overlap, we don’t use recursion.
Instead, we construct optimal subproblems “bottom-up.”
$N_{i,i}$’s are easy, so start with them.
Then do length 2, 3, … subproblems, and so on.
The running time is $O(n^3)$

Algorithm $\text{matrixChain}(S)$:

Input: sequence $S$ of $n$ matrices to be multiplied
Output: number of operations in an optimal paranethization of $S$

for $i \leftarrow 1$ to $n-1$ do
    $N_{i,i} \leftarrow 0$

for $b \leftarrow 1$ to $n-1$ do
    for $i \leftarrow 0$ to $n-b-1$ do
        $j \leftarrow i+b$
        $N_{i,j} \leftarrow +\text{infinity}$
        for $k \leftarrow i$ to $j-1$ do
            $N_{i,j} \leftarrow \min\{N_{i,j} , N_{i,k} +N_{k+1,j} +d_i d_{k+1} d_{j+1}\}$
A Dynamic Programming Algorithm Visualization

- The bottom-up construction fills in the N array by diagonals.
- \( N_{i,j} \) gets values from previous entries in \( i \)-th row and \( j \)-th column.
- Filling in each entry in the N table takes \( O(n) \) time.
- Total run time: \( O(n^3) \)
- Getting actual parenthesization can be done by remembering “k” for each N entry.

\[
N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}
\]
Determining the Run Time

It’s not easy to see that filling in each square takes $O(n)$ time. So, let’s look at this another way

$$(n - 1) + 2(n - 2) + 3(n - 3) + 4(n - 4) + \cdots + (n - 1)(n - (n - 1))$$

$$= \sum_{i=1}^{n-1} i(n - i) = n \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2$$

$$= n \frac{(n - 1)n}{2} - \frac{(n - 1)n(2(n - 1) + 1)}{6}$$

$$= n(n - 1) \left[ \frac{n}{2} - \frac{2n - 1}{6} \right] = n(n - 1) \frac{n + 1}{6}$$

$O(n^3)$
The General 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).
Dynamic Programming:
Telescope Scheduling
Motivation

Large, powerful telescopes are precious resources that are typically oversubscribed by the astronomers who request times to use them.

This high demand for observation times is especially true, for instance, for a space telescope, which could receive thousands of observation requests per month.
Telescope Scheduling Problem

The input to the telescope scheduling problem is a list, $L$, of observation requests, where each request, $i$, consists of the following elements:

- a **requested start time**, $s_i$, which is the moment when a requested observation should begin
- a **finish time**, $f_i$, which is the moment when the observation should finish (assuming it begins at its start time)
- a positive numerical **benefit**, $b_i$, which is an indicator of the scientific gain to be had by performing this observation.

The start and finish times for an observation request are specified by the astronomer requesting the observation; the benefit of a request is determined by an administrator or a review committee.
Telescope Scheduling Problem

To get the benefit, $b_i$, for an observation request, $i$, that observation must be performed by the telescope for the entire time period from the start time, $s_i$, to the finish time, $f_i$.

Thus, two requests, $i$ and $j$, conflict if the time interval $[s_i, f_i]$, intersects the time interval, $[s_j, f_j]$.

Given the list, $L$, of observation requests, the optimization problem is to schedule observation requests in a nonconflicting way so as to maximize the total benefit of the observations that are included in the schedule.
Example

The left and right boundary of each rectangle represent the start and finish times for an observation request. The height of each rectangle represents its benefit. We list each request’s benefit (Priority) on the left. The optimal solution has total benefit 17 = 5 + 5 + 2 + 5.
False Start 1: Brute Force

There is an obvious exponential-time algorithm for solving this problem, of course, which is to consider all possible subsets of L and choose the one that has the highest total benefit without causing any scheduling conflicts.

Implementing this brute-force algorithm would take $O(n2^n)$ time, where $n$ is the number of observation requests.

We can do much better than this, however, by using the dynamic programming technique.
A natural greedy strategy would be to consider the observation requests ordered by nonincreasing benefits, and include each request that doesn’t conflict with any chosen before it.

- This strategy doesn’t lead to an optimal solution, however.

For instance, suppose we had a list containing just 3 requests—one with benefit 100 that conflicts with two nonconflicting requests with benefit 75 each.

- The greedy method would choose the observation with benefit 100, whereas we can achieve a total benefit of 150 by taking the two requests with benefit 75 each.
- So a greedy strategy based on repeatedly choosing a nonconflicting request with maximum benefit won’t work.
The General 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).
Defining Simple Subproblems

A natural way to define subproblems is to consider the observation requests according to some ordering, such as ordered by start times, finish times, or benefits.

- We already saw that ordering by benefits is a false start.
- Start times and finish times are essentially symmetric, so let us order observations by finish times.

\[ B_i = \text{the maximum benefit that can be achieved with the first } i \text{ requests in } L. \]

So, as a boundary condition, we get that \( B_0 = 0 \).
Predecessors

For any request $i$, the set of other requests that conflict with $i$ form a contiguous interval of requests in $L$.

Define the **predecessor** $\text{pred}(i)$, for each request, $i$, then, to be the largest index, $j < i$, such that requests $i$ and $j$ don’t conflict. If there is no such index, then define the predecessor of $i$ to be 0.
Subproblem Optimality

A schedule that achieves the optimal value, $B_i$, either includes observation $i$ or not.

- If the optimal schedule achieving the benefit $B_i$ includes observation $i$, then $B_i = B_{\text{pred}(i)} + b_i$. If this were not the case, then we could get a better benefit by substituting the schedule achieving $B_{\text{pred}(i)}$ for the one we used from among those with indices at most $\text{pred}(i)$.
- On the other hand, if the optimal schedule achieving the benefit $B_i$ does not include observation $i$, then $B_i = B_{i-1}$. If this were not the case, then we could get a better benefit by using the schedule that achieves $B_{i-1}$.

Therefore, we can make the following recursive definition:

$$B_i = \max\{B_{i-1}, B_{\text{pred}(i)} + b_i\}.$$
Subproblem Overlap

- The above definition has subproblem overlap.
- Thus, it is most efficient for us to use memoization when computing $B_i$ values, by storing them in an array, $B$, which is indexed from 0 to $n$.
- Given the ordering of requests by finish times and an array, $P$, so that $P[i] = \text{pred}(i)$, then we can fill in the array, $B$, using the following simple algorithm:

\[
B[0] \leftarrow 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad B[i] \leftarrow \max\{B[i-1], B[P[i]] + b_i\}
\]

After this algorithm completes, the benefit of the optimal solution will be $B[n]$. 
Analysis of the Algorithm

- It is easy to see that the running time of this algorithm is $O(n)$, assuming the list $L$ is ordered by finish times and we are given the predecessor for each request $i$.

- Of course, we can easily sort $L$ by finish times if it is not given to us already sorted according to this ordering.

- To compute the predecessor of each request, note that it is sufficient that we also have the requests in $L$ sorted by start times.
  - In particular, given a listing of $L$ ordered by finish times and another listing, $L'$, ordered by start times, then a merging of these two lists, as in the merge-sort algorithm (Section 8.1), gives us what we want.
  - The predecessor of request $i$ is literally the index of the predecessor in $L$ of the value, $s_i$, in $L'$. 

Telescope Scheduling
Dynamic Programming: Game Strategies

Football signed by President Gerald Ford when playing for University of Michigan. Public domain image.
“Coins in a Line” is a game whose strategy is sometimes asked about during job interviews.

In this game, an even number, $n$, of coins, of various denominations, are placed in a line.

Two players, who we will call Alice and Bob, take turns removing one of the coins from either end of the remaining line of coins.

The player who removes a set of coins with larger total value than the other player wins and gets to keep the money. The loser gets nothing.

Alice’s goal: get the most.
False Start 1: Greedy Method

A natural **greedy** strategy is “always choose the largest-valued available coin.”

But this doesn’t always work:

- [5, 10, 25, 10]: Alice chooses 10
- [5, 10, 25]: Bob chooses 25
- [5, 10]: Alice chooses 10
- [5]: Bob chooses 5

Alice’s total value: 20, Bob’s total value: 30. (Bob wins, Alice loses)
False Start 2: Greedy Method

Another **greedy** strategy is “choose odds or evens, whichever is better.”

Alice can always win with this strategy, but won’t necessarily get the most money.

Example: [1, 3, 6, 3, 1, 3]

Alice’s total value: $9, Bob’s total value: $8.

Alice wins $9, but could have won $10.

How?
The General 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).
Since Alice and Bob can remove coins from either end of the line, an appropriate way to define subproblems is in terms of a range of indices for the coins, assuming they are initially numbered from 1 to n.

Thus, let us define the following indexed parameter:

\[
M_{i,j} = \begin{cases} 
\text{the maximum value of coins taken by Alice, for coins numbered } i \text{ to } j, & \text{assuming Bob plays optimally.} \\
\end{cases}
\]

Therefore, the optimal value for Alice is determined by \(M_{1,n}\).
Subproblem Optimality

Let us assume that the values of the coins are stored in an array, $V$, so that coin 1 is of Value $V[1]$, coin 2 is of Value $V[2]$, and so on.

Note that, given the line of coins from coin $i$ to coin $j$, the choice for Alice at this point is either to take coin $i$ or coin $j$ and thereby gain a coin of value $V[i]$ or $V[j]$.

Once that choice is made, play turns to Bob, who we are assuming is playing optimally.

- We should assume that Bob will make the choice among his possibilities that minimizes the total amount that Alice can get from the coins that remain.
Subproblem Overlap

Alice should choose based on the following:

- If $j = i + 1$, then she should pick the larger of $V[i]$ and $V[j]$, and the game is over.
- Otherwise, if Alice chooses coin $i$, then she gets a total value of
  \[
  \min\{M_{i+1,j-1}, M_{i+2,j}\} + V[i].
  \]
- Otherwise, if Alice chooses coin $j$, then she gets a total value of
  \[
  \min\{M_{i,j-2}, M_{i+1,j-1}\} + V[j].
  \]

That is, we have initial conditions, for $i=1,2,...,n-1$:

\[
M_{i,i+1} = \max\{V[i], V[i+1]\}.
\]

And general equation:

\[
M_{i,j} = \max \left\{ \min\{M_{i+1,j-1}, M_{i+2,j}\} + V[i], \min\{M_{i,j-2}, M_{i+1,j-1}\} + V[j] \right\}.
\]
Analysis of the Algorithm

- We can compute the $M_{i,j}$ values, then, using memoization, by starting with the definitions for the above initial conditions and then computing all the $M_{i,j}$’s where $j - i + 1$ is 4, then for all such values where $j - i + 1$ is 6, and so on.

- Since there are $O(n)$ iterations in this algorithm and each iteration runs in $O(n)$ time, the total time for this algorithm is $O(n^2)$.

- To recover the actual game strategy for Alice (and Bob), we simply need to note for each $M_{i,j}$ whether Alice should choose coin $i$ or coin $j$. 
Dynamic Programming: Longest Common Subsequences
Application: DNA Sequence Alignment

- DNA sequences can be viewed as strings of A, C, G, and T characters, which represent nucleotides.
- Finding the similarities between two DNA sequences is an important computation performed in bioinformatics.
  - For instance, when comparing the DNA of different organisms, such alignments can highlight the locations where those organisms have identical DNA patterns.
Application: DNA Sequence Alignment

Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other.

A brute-force search would take exponential time, but we can do much better using **dynamic programming**.

**Figure 12.1:** Two DNA sequences, $X$ and $Y$, and their alignment in terms of a longest subsequence, $GTCGTCGGAAGCCCGGCGA$, that is common to these two strings.
The General 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).
Subsequences

A **subsequence** of a character string \(x_0x_1x_2...x_{n-1}\) is a string of the form \(x_{i_1}x_{i_2}...x_{i_k}\), where \(i_j < i_{j+1}\).

Not the same as substring!

Example String: ABCDEFGHIJK

- Subsequence: ACEGJIK
- Subsequence: DFGHK
- Not subsequence: DAGH
The Longest Common Subsequence (LCS) Problem

Given two strings $X$ and $Y$, the longest common subsequence (LCS) problem is to find a longest subsequence common to both $X$ and $Y$.

Has applications to DNA similarity testing (alphabet is \{A,C,G,T\}).

Example: ABCDEFG and XZACKDFWGH have ACDFG as a longest common subsequence.
A Poor Approach to the LCS Problem

A Brute-force solution:
- Enumerate all subsequences of X
- Test which ones are also subsequences of Y
- Pick the longest one.

Analysis:
- If X is of length n, then it has $2^n$ subsequences
- This is an exponential-time algorithm!
A Dynamic-Programming Approach to the LCS Problem

Define \( L[i,j] \) to be the length of the longest common subsequence of \( X[0..i] \) and \( Y[0..j] \).

Allow for -1 as an index, so \( L[-1,k] = 0 \) and \( L[k,-1] = 0 \), to indicate that the null part of \( X \) or \( Y \) has no match with the other.

Then we can define \( L[i,j] \) in the general case as follows:
1. If \( x_i = y_j \), then \( L[i,j] = L[i-1,j-1] + 1 \) (we can add this match)
2. If \( x_i \neq y_j \), then \( L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \) (we have no match here)

**Case 1:**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
Y = CGATAATTGAGA \\
L[8,10] = 5
\end{array}
\]

**Case 2:**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
Y = CGATAATTGAG \\
L[9,9] = 6 \\
L[8,10] = 5
\end{array}
\]
An LCS Algorithm

**Algorithm** LCS(X,Y):

**Input:** Strings X and Y with n and m elements, respectively

**Output:** For i = 0,...,n-1, j = 0,...,m-1, the length L[i, j] of a longest string that is a subsequence of both the string X[0..i] = x_0x_1x_2...x_i and the string Y [0.. j] = y_0y_1y_2...y_j

for i =1 to n-1 do
    L[i,-1] = 0
for j =0 to m-1 do
    L[-1,j] = 0
for i =0 to n-1 do
    for j =0 to m-1 do
        if x_i = y_j then
            L[i, j] = L[i-1, j-1] + 1
        else
            L[i, j] = max{L[i-1, j], L[i, j-1]}

return array L
# Visualizing the LCS Algorithm

<table>
<thead>
<tr>
<th>$L$</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

$X = \text{GTTCCCTAATA}$

$Y = \text{CGATAATTGAGA}$

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
Analysis of LCS Algorithm

- We have two nested loops
  - The outer one iterates $n$ times
  - The inner one iterates $m$ times
  - A constant amount of work is done inside each iteration of the inner loop
  - Thus, the total running time is $O(nm)$

- Answer is contained in $L[n,m]$ (and the subsequence can be recovered from the $L$ table).
Dynamic Programming: 0/1 Knapsack
The 0/1 Knapsack Problem

- 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 $W$.
- If we are not allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let $T$ denote the set of items we take
    - Objective: maximize $\sum_{i \in T} b_i$
    - Constraint: $\sum_{i \in T} w_i \leq W$
Example

Given: A set $S$ of $n$ items, with each item $i$ having
- $b_i$ - a positive “benefit”
- $w_i$ - a positive “weight”

Goal: Choose items with maximum total benefit but with weight at most $W$.

Items:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4 in</td>
<td>$20</td>
</tr>
<tr>
<td>2</td>
<td>2 in</td>
<td>$3</td>
</tr>
<tr>
<td>3</td>
<td>2 in</td>
<td>$6</td>
</tr>
<tr>
<td>4</td>
<td>6 in</td>
<td>$25</td>
</tr>
<tr>
<td>5</td>
<td>2 in</td>
<td>$80</td>
</tr>
</tbody>
</table>

“knapsack”

box of width 9 in

Solution:
- item 5 ($80, 2 in)
- item 3 ($6, 2 in)
- item 1 ($20, 4 in)
The General 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).
A 0/1 Knapsack Algorithm, First Attempt

- $S_k$: Set of items numbered 1 to $k$.
- Define $B[k] = \text{best selection from } S_k$.
- Problem: 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 $W = 20$

Best for $S_4$:

Best for $S_5$:
A 0/1 Knapsack Algorithm, Second (Better) Attempt

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

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

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
0/1 Knapsack Algorithm

Recall the definition of $B[k, w]$

Since $B[k, w]$ is defined in terms of $B[k-1, *]$, we can use two arrays of instead of a matrix

Running time: $O(nW)$.

Not a polynomial-time algorithm since $W$ may be large

This is a pseudo-polynomial time algorithm

Algorithm 01Knapsack$(S, W)$:

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 $W$

let $A$ and $B$ be arrays of length $W + 1$

for $w \leftarrow 0$ to $W$ do

$B[w] \leftarrow 0$

for $k \leftarrow 1$ to $n$ do

copy array $B$ into array $A$

for $w \leftarrow w_k$ to $W$ do

if $A[w-w_k]+b_k > A[w]$ then

$B[w] \leftarrow A[w-w_k]+b_k$

return $B[W]$