Dynamic Programming

- Give a solution of a problem using smaller sub-problems where all the possible sub-problems are determined in advance
- Useful when the same sub-problems show up again and again in the solution

A simple case: Computing Fibonacci Numbers

- Recall \( F_n = F_{n-1} + F_{n-2} \) and \( F_0 = 0, F_1 = 1 \)
- Recursive algorithm:
  - \( \text{Fibo}(n) \)
    - if \( n = 0 \) then return(0)
    - else if \( n = 1 \) then return(1)
    - else return(\( \text{Fibo}(n-1) + \text{Fibo}(n-2) \))

Call tree - start

Memoization (Caching)

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed
- Convert memoized algorithm from a recursive one to an iterative one
Fibonacci Dynamic Programming Version

FiboDP(n):
F\[0\] ← 0
F\[1\] ← 1
for i=2 to n do
  F\[i\] ← F\[i-1\]+F\[i-2\]
endfor
return(F\[n\])

Fibonacci: Space-Saving Dynamic Programming

FiboDP(n):
prev ← 0
curr ← 1
for i=2 to n do
  temp ← curr
  curr ← curr + prev
  prev ← temp
endfor
return(curr)

Dynamic Programming

Useful when
- same recursive sub-problems occur repeatedly
- Can anticipate the parameters of these recursive calls
- The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved
- principle of optimality
  “Optimal solutions to the sub-problems suffice for optimal solution to the whole problem”

Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different values of parameters in the recursive calls is “small”
  - e.g., bounded by a low-degree polynomial
  - Can use memoization
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

Weighted Interval Scheduling

- Same problem as interval scheduling except that each request \(i\) also has an associated value or weight \(w_i\)
- \(w_i\) might be
  - amount of money we get from renting out the resource for that time period
  - amount of time the resource is being used
- Goal: Find compatible subset \(S\) of requests with maximum total weight

Greedy Algorithms for Weighted Interval Scheduling?

- No criterion seems to work
  - Earliest start time \(s_i\)
    - Doesn’t work
  - Shortest request time \(f_i-s_i\)
    - Doesn’t work
  - Fewest conflicts
    - Doesn’t work
  - Earliest finish time \(f_i\)
    - Doesn’t work
  - Largest weight \(w_i\)
    - Doesn’t work
Towards Dynamic Programming: Step 1 – A Recursive Algorithm

Suppose that like ordinary interval scheduling we have first sorted the requests by finish time \( f \), so \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Say request \( i \) comes before request \( j \) if \( i < j \).

For any request \( j \) let \( p(j) \) be

- the largest-numbered request before \( j \) that is compatible with \( j \)
- or 0 if no such request exists

Therefore \( \{1, \ldots, p(j)\} \) is precisely the set of requests before \( j \) that are compatible with \( j \).

Towards Dynamic Programming: Step 1 – A Recursive Algorithm

Two cases depending on whether an optimal solution \( O \) includes request \( n \)

- If it \textbf{does} include request \( n \) then all other requests in \( O \) must be contained in \( \{1, \ldots, p(n)\} \)
- Not only that!
  - Any set of requests in \( \{1, \ldots, p(n)\} \) will be compatible with request \( n \)
  - So in this case the optimal solution \( O \) must contain an optimal solution for \( \{1, \ldots, p(n)\} \)
  - “Principle of Optimality”

Towards Dynamic Programming: Step 2 – Small # of parameters

\[
\text{ComputeOpt}(n) \text{ can take exponential time in the worst case}
\]

- \( 2^n \) calls if \( p(i) = i-1 \) for every \( i \)
- There are only \( n \) possible parameters to \text{ComputeOpt}
- Store these answers in an array \( \text{OPT}[n] \) and only recompute when necessary
- \text{Memoization}
- Initialize \( \text{OPT}[i] = 0 \) for \( i = 1, \ldots, n \)
Dynamic Programming: Step 2 – Memoization

```plaintext
ComputeOpt(n)
if n=0 then return(0)
else
  u ← MComputeOpt(p[n])
  v ← MComputeOpt(n-1)
  if w[n]+u > v then
    return(w[n]+u)
  else return(v)
endif
```

MComputeOpt(n)
if OPT[n]=0 then
  v ← ComputeOpt(n)
  OPT[n] ← v
else
  return(OPT[n])
endif
```

Dynamic Programming Step 3: Iterative Solution

The recursive calls for parameter n have parameter values i that are < n

```plaintext
IterativeComputeOpt(n)
array OPT[0..n], Used[1..n]
OPT[0] ← 0
for i=1 to n
  if w[i]+OPT[p[i]] > OPT[i-1] then
    OPT[i] ← w[i]+OPT[p[i]]
    Used[i] ← 1
  else
    OPT[i] ← OPT[i-1]
    Used[i] ← 0
  endif
endfor
i ← n
S ← Ø
while i > 0 do
  if Used[i]=1 then
    S ← S È {i}
  else
    i ← i-1
  endif
endwhile
```

Example
```
<table>
<thead>
<tr>
<th>i</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>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>p</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>w</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Example
```
<table>
<thead>
<tr>
<th>i</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>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>p</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>w</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>
```
Example

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>s</td>
<td>t</td>
<td>w</td>
<td>p(i)</td>
<td>OPT(i)</td>
<td>Used(i)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>11</td>
<td>12</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

S=(9,7,2)

Segmented Least Squares

- Least Squares
  - Given a set \( P \) of \( n \) points in the plane \( p_1=(x_1,y_1),\ldots,p_n=(x_n,y_n) \) with \( x_1<\ldots<x_n \) determine a line \( L \) given by \( y=ax+b \) that optimizes the totaled ‘squared error’
    - \( \text{Error}(L,P)=\sum(y_i-ax_i-b)^2 \)
  - A classic problem in statistics
  - Optimal solution is known (see text)
  - Call this line \( (P) \) and its error \( \text{error}(P) \)

Least Squares

- What if data seems to follow a piece-wise linear model?
Segmented Least Squares

- What if data seems to follow a piece-wise linear model?
- Number of pieces to choose is not obvious
- If we chose \( n-1 \) pieces we could fit with 0 error
  - Not fair
- Add a penalty of \( C \) times the number of pieces to the error to get a total penalty
- How do we compute a solution with the smallest possible total penalty?

Recursive idea

- If we knew the point \( p_j \) where the last line segment began then we could solve the problem optimally for points \( p_1, \ldots, p_j \) and combine that with the last segment to get a global optimal solution
- Let \( OPT(i) \) be the optimal penalty for points \( \{p_1, \ldots, p_i\} \)
- Total penalty for this solution would be \( \text{Error}([p_j, \ldots, p_n]) + C + OPT(j-1) \)

Dynamic Programming Solution

```c
SegmentedLeastSquares(n)
array OPT[0..n], Begin[1..n]
OPT[0] ← 0
for i = 1 to n do
  OPT[i] ← Error([p_1, ..., p_i]) + C
  for j = 2 to i do
    e ← Error([p_j, ..., p_i]) + C + OPT[j-1]
    if e < OPT[i] then
      OPT[i] ← e
      Begin[i] ← j
  endfor
endfor
return OPT[n]
```

Knapsack (Subset-Sum) Problem

- Given:
  - integer \( W \) (knapsack size)
  - \( n \) object sizes \( x_1, x_2, \ldots, x_n \)
- Find:
  - Subset \( S \) of \( \{1, \ldots, n\} \) such that \( \sum_{i \in S} x_i \leq W \) but \( \sum_{i \in S} x_i \) is as large as possible
Recursive Algorithm

- Let \( K(n, W) \) denote the problem to solve for \( W \) and \( x_1, x_2, \ldots, x_n \).
- For \( n > 0 \),
  - The optimal solution for \( K(n, W) \) is the better of the optimal solution for either \( K(n-1, W) \) or \( x_n + K(n-1, W-x_n) \).
- For \( n = 0 \),
  - \( K(0, W) \) has a trivial solution of an empty set \( S \) with weight 0.

Recursive calls

- Recursive calls on list ..., 3, 4, 7

Common Sub-problems

- Only sub-problems are \( K(i, w) \) for
  - \( i = 0, 1, \ldots, n \)
  - \( w = 0, 1, \ldots, W \)
- Dynamic programming solution
  - Table entry for each \( K(i, w) \)
    - \( OPT \) - value of optimal solution for first \( i \) objects and weight \( w \)
    - \( belong \) flag - is \( x_i \) a part of this solution?
  - Initialize \( OPT[0, w] \) for \( w = 0, \ldots, W \)
  - Compute all \( OPT[i, \cdot] \) from \( OPT[i-1, \cdot] \) for \( i = 0 \)

Dynamic Knapsack Algorithm

\[
\begin{aligned}
&\text{for } w = 0 \text{ to } W: \quad OPT[0, w] \leftarrow 0; \\
&\text{end for} \\
&\text{for } i = 1 \text{ to } n \\
&\quad \text{for } w = 0 \text{ to } W \\
&\quad\quad OPT[i, w] \leftarrow \max \{ OPT[i-1, w], OPT[i-1, w-x_i] \} \\
&\quad\quad \text{if } w \geq x_i \text{ then} \\
&\quad\quad\quad \text{val} \leftarrow x_i + OPT[w-x_i] \\
&\quad\quad\quad \text{if } \text{val} \geq OPT[i, w] \text{ then} \\
&\quad\quad\quad\quad OPT[i, w] \leftarrow \text{val} \\
&\quad\quad\quad\quad belong[i, w] \leftarrow 1 \\
&\quad\quad \text{end for} \\
&\quad \text{end for} \\
&\text{end for} \\
&\text{return } OPT[n, W]
\end{aligned}
\]

Time \( O(nW) \)

Sample execution on 2, 3, 4, 7 with \( K = 15 \)

Saving Space

- To compute the value \( OPT \) of the solution only need to keep the last two rows of \( OPT \) at each step
- What about determining the set \( S \)?
  - Follow the \( belong \) flags \( O(n) \) time
  - What about space?
Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different values of parameters in the recursive algorithm is “small”
  - e.g., bounded by a low-degree polynomial
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

Sequence Alignment: Edit Distance

- Given:
  - Two strings of characters $A=a_1, a_2, ..., a_n$ and $B=b_1, b_2, ..., b_m$
- Find:
  - The minimum number of edit steps needed to transform $A$ into $B$ where an edit can be:
    - insert a single character
    - delete a single character
    - substitute one character by another

Sequence Alignment vs Edit Distance

- Sequence Alignment
  - Insert corresponds to aligning with a “−” in the first string
    - Cost $\delta$ (in our case 1)
  - Delete corresponds to aligning with a “−” in the second string
    - Cost $\delta$ (in our case 1)
  - Replacement of an a by a b corresponds to a mismatch
    - Cost $\delta_{ab}$ (in our case 1 if $a \neq b$ and 0 if $a=b$)
- In Computational Biology this alignment algorithm is attributed to Smith & Waterman

Applications

- "diff" utility – where do two files differ
- Version control & patch distribution – save/send only changes
- Molecular biology
  - Similar sequences often have similar origin and function
  - Similarity often recognizable despite millions or billions of years of evolutionary divergence

Recursive Solution

- Sub-problems: Edit distance problems for all prefixes of $A$ and $B$ that don’t include all of both $A$ and $B$
- Let $D(i,j)$ be the number of edits required to transform $a_1, a_2, ..., a_i$ into $b_1, b_2, ..., b_j$
- Clearly $D(0,0)=0$
Computing $D(n,m)$

- Imagine how best sequence handles the last characters $a_n$ and $b_m$
- If best sequence of operations
  - deletes $a_n$ then $D(n,m) = D(n-1,m) + 1$
  - inserts $b_m$ then $D(n,m) = D(n,m-1) + 1$
  - replaces $a_n$ by $b_m$ then $D(n,m) = D(n-1,m-1) + 1$
  - matches $a_n$ and $b_m$ then $D(n,m) = D(n-1,m-1)$

Recursive algorithm $D(n,m)$

```python
if n=0 then
  return m
else
  m=0 then
    return n
else
  if $a_n$ is $b_m$ then
    replace-cost ← 0
  else
    replace-cost ← 1
  endif
  return \min (D(n-1,m) + 1, D(n,m-1) + 1, D(n-1,m-1) + replace-cost)
end if
```

Dynamic Programming

```plaintext
for j = 0 to m:
  D(0, j) ← j; endfor
for i = 1 to n:
  D(i, 0) ← i; endfor
for j = 1 to m:
  if $a_i$ is $b_j$ then
    replace-cost ← 0
  else
    replace-cost ← 1
  endif
  D(i, j) ← \min (D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + replace-cost)
endfor
```

Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</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>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Example run with AGACATTG and GAGTTA

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<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>3</td>
<td>3</td>
<td>3</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>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Example run with AGACATTG and GAGTTA

Reading off the operations
- Follow the sequence and use each color of arrow to tell you what operation was performed.
- From the operations can derive an optimal alignment

AGACATTG

GAGTTA

Saving Space
- To compute the distance values we only need the last two rows (or columns)
  - $O(\min(m, n))$ space
- To compute the alignment/sequence of operations
  - seem to need to store all $O(mn)$ pointers/arrow colors
- Nifty divide and conquer variant that allows one to do this in $O(\min(m, n))$ space and retain $O(mn)$ time
  - In practice the algorithm is usually run on smaller chunks of a large string, e.g., $m$ and $n$ are lengths of genes so a few thousand characters
  - Researchers want all alignments that are close to optimal
  - Basic algorithm is run since the whole table of pointers ($2$ bits each) will fit in RAM
- Ideas are neat, though
Saving space

- Alignment corresponds to a path through the table from lower right to upper left
- Must pass through the middle column
- Recursively compute the entries for the middle column from the left
- If we knew the cost of completing each then we could figure out where the path crossed

Solution
- Recursively calculate the right half costs for each entry in this column using alignments starting at the other ends of the two input strings!
- Can reuse the storage on the left when solving the right hand problem

Shortest paths with negative cost edges (Bellman-Ford)

- Observe that the recursion for $\text{Cost}(s,t,i)$ doesn’t change
- Only store an entry for each $v$ and $i$
  - Termed $\text{OPT}(v,i)$ in the text
- Also observe that to compute $\text{OPT}(*,i)$ we only need $\text{OPT}(*,i-1)$
  - Can store a current and previous copy in $O(n)$ space.

Bellman-Ford

$\text{ShortestPath}(G,s,t)$

for all $v \in V$
- $\text{OPT}[v] \leftarrow \infty$
- $\text{OPT}[t] \leftarrow 0$

for $i=1$ to $n-1$ do
  for all $v \in V$ do
    $\text{OPT}[v] \leftarrow \min(\text{OPT}[v], c_{vw} + \text{OPT}[w])$
  for all $v \in V$ do
    $\text{OPT}[v] \leftarrow \min(\text{OPT}[v], \text{OPT}[v])$
return $\text{OPT}[s]$

$O(mn)$ time

Negative cycles

- Claim: There is a negative-cost cycle that can reach $t$ iff for some vertex $v \in V$, $\text{Cost}(v,t,n) < \text{Cost}(v,t,n-1)$
- Proof:
  - We already know that if there aren’t any then we only need paths of length up to $n-1$
  - For the other direction
    - The recurrence computes $\text{Cost}(v,t,i)$ correctly for any number of hops $i$
    - The recurrence reaches a fixed point if
**Last details**

- Can run algorithm and stop early if the OPT and OPT' arrays are ever equal.
- Even better, one can update only neighbors v of vertices w with OPT[w]=OPT'[w]
- Can store a successor pointer when we compute OPT.
- Homework assignment

By running for step n we can find some vertex v on a negative cycle and use the successor pointers to find the cycle.
Bellman-Ford with a DAG

Edges only go from lower to higher-numbered vertices
- Update distances in reverse order of topological sort
- Only one pass through vertices required
- \(O(n+m)\) time