Dynamic Programming

• Useful when
  – Same recursive sub-problems occur repeatedly
  – Can anticipate them
  – Can find solution to whole problem without knowing internal details of sub-problem solutions
    • “principle of optimality”
List partition problem

- **Given:** a sequence of $n$ positive integers $s_1, \ldots, s_n$ and a positive integer $k$

- **Find:** a partition of the list into up to $k$ blocks:

$$s_1, \ldots, s_{i_1} | s_{i_1+1} \ldots s_{i_2} | s_{i_2+1} \ldots s_{i_{k-1}} | s_{i_{k-1}+1} \ldots s_n$$

so that the sum of the numbers in the largest block is as small as possible.

i.e., find spots for up to $k-1$ dividers.
Greedy approach

- Ideal size would be $P = \sum_{i=1}^{n} s_i/k$

- Greedy: walk along until what you have so far adds up to $\geq P$ then insert a divider

- Problem: it may not exact (or correct)

  100 200 400 500 900 700 600 800 600, k=3
  - sum is 4800 so size must be at least 1600.
  - Greedy? Best?
Recursive solution

- Try all possible values for the position of the last divider
- For each position of this last divider
  - there are $k-2$ other dividers that must divide the list of numbers prior to the last divider as evenly as possible
    - $s_1, \ldots, s_{i_1} | s_{i_1+1} \ldots s_{i_2} | s_{i_2+1} \ldots s_{i_{k-1}} | s_{i_{k-1}+1} \ldots s_n$
  - recursive sub-problem of the same type
Recursive idea

- Let $M[n,k]$ the smallest cost (size of largest block) of any partition of the $n$ into $k$ pieces.

- If best position for last divider lies between the $i^{th}$ and $i+1^{st}$ then
  
  $$M[n,k] = \max \left( M[i,k-1], \sum_{j=i+1}^{n} s_j \right)$$

- In general
  
  $$M[n,k] = \min_{i<n} \max \left( M[i,k-1], \sum_{j=i+1}^{n} s_j \right)$$

- Base case(s)?
Time-saving - prefix sums

- Computing the costs of the blocks may be expensive and involved repeated work
- Idea: Pre-compute prefix sums
- Length of block
  \[ s_{i+1} + \ldots + s_j \]
  is just
  \[ p[j] - p[i] \]
- Cost: \( n \) additions

\[
\begin{align*}
p[1] &= s_1 \\
p[2] &= s_1 + s_2 \\
p[3] &= s_1 + s_2 + s_3 \\
& \quad \quad \quad \vdots \\
p[n] &= s_1 + s_2 + \ldots + s_n
\end{align*}
\]
Linear Partition Algorithm

Partition(S,k):

\[p[0] \leftarrow 0; \text{ for } i=1 \text{ to } n \text{ do } p[i] \leftarrow p[i-1]+s_i\]

\[\text{ for } i=1 \text{ to } n \text{ do } M[i,1] \leftarrow p[i]\]

\[\text{ for } j=1 \text{ to } k \text{ do } M[1,j] \leftarrow s_1\]

\[\text{ for } i=2 \text{ to } n \text{ do }\]

\[\text{ for } j=2 \text{ to } k \text{ do }\]

\[M[i,j] \leftarrow \min_{pos<i}\{\max(M[pos,j-1], p[i]-p[pos])}\]

\[\sum_{j=pos+1}^{i} s_j\]
Trace-Back: Finding Solns

• Above gives value of best solution
• Q: How do you find it?

• A: work backwards from answer
Linear Partition Algorithm

Partition(S,k):

\[ p[0] \leftarrow 0; \text{ for } i=1 \text{ to } n \text{ do } p[i] \leftarrow p[i-1] + s_i \]

\[ \text{ for } i=1 \text{ to } n \text{ do } M[i,1] \leftarrow p[i] \]

\[ \text{ for } j=1 \text{ to } k \text{ do } M[1,j] \leftarrow s_1 \]

\[ \text{ for } i=2 \text{ to } n \text{ do } \]

\[ \text{ for } j=2 \text{ to } k \text{ do } \]

\[ M[i,j] \leftarrow \min_{pos<i} \{ \max(M[pos,j-1], p[i]-p[pos]) \} \]

\[ D[i,j] \leftarrow \text{value of pos where min is achieved} \]
Linear Partition Algorithm

Partition(S,k):
p[0]←0; for i=1 to n do p[i] ←p[i-1]+s_i
  for i=1 to n do M[i,1] ←p[i]
  for j=1 to k do M[1,j] ← s_1
  for i=2 to n do
    for j=2 to k do
      M[i,j]←∞
      for pos=1 to i-1 do
        s←max(M[pos,j-1], p[i]-p[pos])
        if M[i,j]>s then
          M[i,j] ←s ; D[i,j] ←pos
Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>900</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>700</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>2100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>700</td>
<td>2800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>3400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>4200</td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>4800</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>300</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>400</td>
<td>700</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>500</td>
<td>1200</td>
<td>700</td>
<td>500</td>
</tr>
<tr>
<td>900</td>
<td>2100</td>
<td>1200</td>
<td>900</td>
</tr>
<tr>
<td>700</td>
<td>2800</td>
<td>1600</td>
<td>1400</td>
</tr>
<tr>
<td>600</td>
<td>3400</td>
<td>2100</td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>4200</td>
<td>2100</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>4800</td>
<td>2700</td>
<td></td>
</tr>
</tbody>
</table>
Exercises

• Finish example
• Make up another example & try it
• Figure out from example(s) where the dividers go
• Write an algorithm that, based on the M & D matrices, figures out where the dividers go
Goals: Skills to learn

• Recognize when dynamic programming is a plausible approach
  – E.g., recursive formulation, repeated subproblems, Global opt depends on opt subsolution, but not details thereof.

• Understand the logic of the correctness of the method from the recurrence & vice versa

• Construct D.P. algorithms for new problems you see