Problems

A computational problem is associated with a function from instances to feasible solutions.

For example, in satisfiability problems, instances are Boolean formulas, and solutions are satisfying assignments.

A Boolean formula is an expression like

\[(x_{25} \land x_{12}) \lor \neg(x_{70} \lor (\neg x_{3} \land x_{34}))\]

built up from the elements \(x_i\) called propositional variables, a finite set of connectives (usually including \(\land\), \(\lor\) and \(\neg\)) and brackets.

An assignment of the truth-values true and false to the variables is satisfying if it makes the formula evaluate to true.
A computational problem requires us to take as input an instance, and compute some information about the set of feasible solutions associated with the instance.

A decision problem (also known as an existence problem) asks the following question:

*Is there any feasible solution for the given instance?*

For example, the decision problem SAT is defined as follows

**Instance:** A Boolean formula $F$

**Question:** Does $F$ have a satisfying assignment?

Several other types of problems are definable in this setting.

1. In a construction problem the goal is to find a feasible solution for the given instance.
2. In a listing problem the goal is to list all feasible solutions for the given instance.
3. In an optimisation problem we associate a numeric value with every pair $(x, y)$ containing a problem instance $x$ and a feasible solution $y$. The goal, given an instance $x$, is to find a solution $y$ for which the value is optimised. That is, the value should be either as large as possible for the given instance $x$ (in a maximisation problem) or as small as possible (in a minimisation problem).
4. In a counting problem, the goal is to determine the number of feasible solutions associated with the given instance.

Another type of decision problem (which will be referred to as the membership problem) answers the question:

*Is some set of data $Y$ a feasible solution for the given instance?*

**Review questions**

1. Define a natural counting problem associated with Boolean formulae.
2. Define a maximisation problem associated with Boolean formulae.
Example

Let \( S \) be a string of text representing a query (e.g. \( S \equiv \) “Efficient algorithms” or “Liverpool players”) and \( \mathcal{W} = \{ W_1, W_2, \ldots \} \) be the collection of all web pages indexed by a particular search engine.

The web search problem \((S, \mathcal{W})\) is that of retrieving all web pages \( W_i \) containing the string \( S \).

It is a listing problem.

Greedy Algorithms

We consider optimisation problems. Algorithms for optimisation problems typically go through a sequence of steps, with a set of choices at each step.

A greedy algorithm is a process that always makes the choice that looks best at the moment.

Greedy algorithms are natural. In a few cases they produce an optimal solution.

We will look at one simple example and we will try to understand why it works.

We will consider more examples later.

Activity-selection problem

We are given a set \( S \) of proposed activities that wish to use a resource. The resource can only be used for one activity at a time.

Each activity \( A \in S \) is defined by a pair consisting of a start time \( s(A) \) and a finish time \( f(A) \). The start time \( s(A) \) is a non-negative number, and the finish time \( f(A) \) is larger than \( s(A) \).

If selected, activity \( A \) takes place during the time interval \([s(A), f(A))\).

Two activities \( A \) and \( A' \) are compatible if \( s(A) \geq f(A') \) or \( s(A') \geq f(A) \).

The activity-selection problem is to select the maximum number of mutually compatible activities.

An optimisation problem \( P \) is defined by four components \((I, S, v, \text{goal})\) where:

(1) \( I \) is the set of the instances of \( P \).

(2) For each \( x \in I \), \( S(x) \) is the set of feasible solutions associated with \( x \).

(3) For each \( x \in I \) and each \( y \in S(x) \), \( v(x, y) \) is a positive integer, which is the value of solution \( y \) for instance \( x \).

(4) \text{goal} \in \{\text{max}, \text{min}\} \) is the optimisation criterion and tells if the problem \( P \) is a maximisation or a minimisation problem.
Open issues

How well can we do?

In a real life situation, there may be hundreds of activities. Suppose that our program, when run on a particular input, returns 50 activities. Is this best-possible? Is it good enough?

We would like to be able argue that our program is “certified” to produce the best possible answer. That is, we’d like to have a mathematical proof that the program returns the best-possible answer.

Example of Real Life Application

“How dependent web browsing”

The access speeds to particular sites on the World Wide Web can vary depending on the time of access.

Access rates for USA and Asia over 24 hours. (Derived from data posted by the Anderson News Network

www.internettrafficreport.com)
A single central computer (e.g. a search engine server) collects all the information stored in a certain number of web documents, located at various sites.

The information is gathered by scheduling a number of consecutive client/server TCP connections with the required web sites.

We assume that the loading time of any particular page from any site may be different at different times, e.g. the access to the page is much slower in peak hours than in off-peak hours.

Having a list of pages to be collected along with some information about the access time at any given instant, the goal is to download as many pages as possible.

Typical input for this problem can be a sequence of tables like the following (times are GMT), one for each remote web site.

<table>
<thead>
<tr>
<th>Site</th>
<th>t2</th>
<th>t24</th>
<th>t25</th>
<th>t1</th>
<th>t4</th>
<th>t3</th>
<th><a href="http://www.one.com">www.one.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.30</td>
<td>11.30</td>
<td>12.00</td>
<td>0.00</td>
<td>1.30</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

It is an optimisation problem.

The set of instances coincides with the set of all possible groups of activities (in tabular form).

A solution is a set of compatible activities.

The value of a solution is its size (or cardinality), and we are seeking a maximum cardinality solution.

This is like the activity selection problem if we make two simplifying assumptions: (1) we keep only the best access period for each web site. (2) we assume that the download happens only once (rather than repeatedly) and the goal is to collect as many sites as possible.

Greedy algorithm for activity selection

1. Input $S$, the set of available activities.
2. Choose the activity $A \in S$ with the earliest finish time.
3. Form a sub-problem $S'$ from $S$ by removing $A$ and removing any activities that are incompatible with $A$.
4. Recursively choose a set of activities $X'$ from $S'$.
5. Output the activity $A$ together with the activities in $X'$. 


Greedy algorithm for activity selection

**GREEDY-ACTIVITY-SELECTOR (S)**

If $S = \emptyset$ Return $\emptyset$

Else

\begin{align*}
  A &\leftarrow \text{first activity in list } S \\
  \text{For every other } A' \in S &\text{ } \\
  \text{If } f(A') < f(A) &\text{ } \\
  A &\leftarrow A' \\
  S' &\leftarrow \emptyset \\
  \text{For every } A' \in S &\text{ } \\
  \text{If } s(A) \geq f(A') \text{ or } s(A') \geq f(A) &\text{ } \\
  S' &\leftarrow S' \cup \{A'\} \\
  X' &\leftarrow \text{GREEDY-ACTIVITY-SELECTOR } (S') \\
  X &\leftarrow \{A\} \cup X' \\
  \text{Return } X
\end{align*}

Example

Let's simulate the algorithm.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
<th>$A_8$</th>
<th>$A_9$</th>
<th>$A_{10}$</th>
<th>$A_{11}$</th>
<th>$A_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(A)$</td>
<td>44</td>
<td>37</td>
<td>83</td>
<td>27</td>
<td>49</td>
<td>16</td>
<td>44</td>
<td>44</td>
<td>58</td>
<td>27</td>
<td>58</td>
<td>27</td>
</tr>
<tr>
<td>$f(A)$</td>
<td>86</td>
<td>25</td>
<td>96</td>
<td>89</td>
<td>84</td>
<td>62</td>
<td>17</td>
<td>70</td>
<td>84</td>
<td>94</td>
<td>79</td>
<td>57</td>
</tr>
</tbody>
</table>

$A = A_7$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$A_1$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_8$</th>
<th>$A_9$</th>
<th>$A_{10}$</th>
<th>$A_{11}$</th>
<th>$A_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(A)$</td>
<td>44</td>
<td>37</td>
<td>83</td>
<td>27</td>
<td>49</td>
<td>44</td>
<td>44</td>
<td>58</td>
<td>27</td>
<td>58</td>
</tr>
<tr>
<td>$f(A)$</td>
<td>86</td>
<td>96</td>
<td>89</td>
<td>84</td>
<td>62</td>
<td>70</td>
<td>84</td>
<td>94</td>
<td>79</td>
<td>57</td>
</tr>
</tbody>
</table>

$A = A_{12}$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$A_4$</th>
<th>$A_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(A)$</td>
<td>83</td>
<td>58</td>
</tr>
<tr>
<td>$f(A)$</td>
<td>89</td>
<td>94</td>
</tr>
</tbody>
</table>

Select $A_{12}$ and the following.

Continuing the Simulation

$A = A_7$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$A_1$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_8$</th>
<th>$A_9$</th>
<th>$A_{10}$</th>
<th>$A_{11}$</th>
<th>$A_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(A)$</td>
<td>44</td>
<td>37</td>
<td>83</td>
<td>27</td>
<td>49</td>
<td>44</td>
<td>44</td>
<td>58</td>
<td>27</td>
<td>58</td>
</tr>
<tr>
<td>$f(A)$</td>
<td>86</td>
<td>96</td>
<td>89</td>
<td>84</td>
<td>62</td>
<td>70</td>
<td>84</td>
<td>94</td>
<td>79</td>
<td>57</td>
</tr>
</tbody>
</table>

Select $A_7$ and the following.

Implementation

Represent $S$ with a data structure such as a linked list. Each list item corresponds to an activity $A$ which has associated data $s(A)$ and $f(A)$.
Continuing the Simulation

A

\[
\begin{array}{c|cc}
A & A_4 & A_{10} \\
\hline
s(A) & 83 & 58 \\
f(A) & 89 & 94 \\
\end{array}
\]

\( A = A_4 \)

Select \( A_4 \) and (nothing else...)

Another example: Time-dependent web browsing

\[(\text{simplified version})\]

\[
\begin{array}{c|cccccccc}
s(A) & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
9.15 & 7.42 & 10.00 & 11.54 & 9.17 & 9.47 & 7.34 & 8.16 & 8.36 \\
10.45 & 11.53 & 11.05 & 10.56 & 12.30 & 12.16 \\
\end{array}
\]

Our algorithm returned the following solution

\[
\begin{array}{c|cccccccc}
A & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & A_8 & A_9 & A_{10} & A_{11} & A_{12} \\
\hline
s(A) & 44 & 7 & 37 & 83 & 27 & 49 & 16 & 44 & 44 & 58 & 27 & 26 \\
f(A) & 86 & 25 & 96 & 89 & 84 & 62 & 17 & 70 & 84 & 94 & 79 & 57 \\
output? & X & X & X \\
\end{array}
\]

Time complexity

Claim. The time complexity of \textsc{Greedy-Activity-Selector} is \( O(n^2) \), where \( n = |S| \).

- Choosing \( A \) takes \( O(n) \) time.
- Constructing \( S' \) takes \( O(n) \) time.
- The rest of the algorithm takes \( O(1) \) time, except for the recursive call on \( S' \).
- But \( |S'| \leq n - 1 \).

\[
T(n) = cn + T(n-1)
\]
\[
= cn + c(n-1) + T(n-2)
\]
\[
= \ldots
\]
\[
= c(n + (n-1) + (n-2) + \cdots + 0)
\]
Digression:

\[1 + 2 + \cdots + n = ?\]

\[1 + 2 + \cdots + n = \frac{n(n + 1)}{2}\]

How would you prove it?

How can we speed up the algorithm?

**GREEDY-ACTIVITY-SELECTOR** \((S)\)

- If \(S = \emptyset\) Return \(\emptyset\)
- Else
  - \(A \leftarrow\) first activity in list \(S\)
  - For every other \(A' \in S\)
    - If \(f(A') < f(A)\)
      - \(A \leftarrow A'\)
    - \(S' \leftarrow \emptyset\)
  - For every \(A' \in S\)
    - If \(s(A) \geq f(A')\) or \(s(A') \geq f(A)\)
      - \(S' \leftarrow S' \cup \{A'\}\)
  - \(X' \leftarrow\) GREEDY-ACTIVITY-SELECTOR \((S')\)
  - \(X \leftarrow \{A\} \cup X'\)
  - Return \(X\)

Sort the items in \(S\) in order of increasing finishing time. Then we can always take \(A\) to be the first element of \(S\).

If \(S\) is sorted in order of increasing finishing time

**GREEDY-ACTIVITY-SELECTOR** \((S)\)

- If \(S = \emptyset\) Return \(\emptyset\)
- Else
  - \(A \leftarrow\) first activity in list \(S\)
  - \(S' \leftarrow \emptyset\)
  - For every \(A' \in S\)
    - If \(s(A) \geq f(A')\) or \(s(A') \geq f(A)\)
      - \(S' \leftarrow S' \cup \{A'\}\)
  - \(X' \leftarrow\) GREEDY-ACTIVITY-SELECTOR \((S')\)
  - \(X \leftarrow \{A\} \cup X'\)
  - Return \(X\)

\(A'\) is incompatible with \(A\) unless it starts after \(A\) finishes.

When we construct \(S'\) we throw out the first few elements of \(S\) until we finally reach an activity \(A'\) that is compatible with \(A\). It turns out that, we can just take \(S'\) to be \(A'\) and all items after it in \(S\).
So the algorithm can be written in such a way that it takes $O(n \log n)$ time, for sorting $S$, followed by $O(n)$ time for the greedy algorithm.

The activity-selection problem has two properties

The **Greedy Choice** Property: Let $S$ be a set of activities and let $A$ be an activity in $S$ with earliest finish time. There is an optimal selection of activities $X$ for $S$ that contains $A$.

The **Recursive** Property: An optimal selection of activities for $S$ that contains $A$ can be found from any optimal solution of the smaller problem instance $S'$: In particular, if $X''$ is an optimal solution for $S'$ then $X'' \cup \{A\}$ is best amongst feasible solutions for $S$ that contain $A$.

**Proof of Correctness**

Proof by induction on $n$, the number of activities in $S$.

**Base cases:** $n = 0$ or $n = 1$.

**Inductive step.** Assuming that the algorithm is optimal with inputs of size at most $n - 1$, we must prove that it returns an optimal solution with an input of size $n$.

Let $S$ be an instance with $|S| = n$. Let $A$ be the activity chosen by the algorithm. By the **Greedy Choice** property, there is an optimal solution containing $A$. By the **Recursive** property, we can construct an optimal solution for $S$ by combining $A$ with an optimal solution $X'$ for $S'$. But by induction, the algorithm does return an optimal solution $X'$ for $S'$.
Proving the Greedy Choice property

The **Greedy Choice** Property: Let $S$ be a set of activities and let $A$ be an activity in $S$ with earliest finish time. There is an optimal selection of activities $X$ for $S$ that contains $A$.

**Proof:**

Suppose $X \subseteq S$ is an optimal solution. Suppose $A'$ is the activity in $X$ with the smallest finish time. If $A = A'$ we are done (because we have found an optimal selection containing $A$). Otherwise, we could replace $A'$ with $A$ in $X$ and obtain another solution $X'$ with the same number of activities as $X$. Then $X'$ is an optimal solution containing $A$.

General Properties of a recursive greedy solution

A simple recursive greedy algorithm produces an optimal answer if the following properties are true.

The **Greedy Choice** Property: For every instance, there is an optimal solution consistent with the first greedy choice.

The **Recursive Property**: For every instance $S$ of the problem there is a smaller instance $S'$ such that, using any optimal solution to $S'$, one obtains a best-possible solution for $S$ amongst all solutions that are consistent with the greedy choice.

Proving the Recursive Property

Suppose that $X''$ is an optimal solution to $S'$. We have to show that $X'' \cup \{A\}$ has as many activities as possible, amongst feasible solutions for $S$ that contain $A$.

Suppose for contradiction that there was some better solution $\{A\} \cup Y'$ for $S$. Then $Y'$ would be a better solution for $S'$ than the supposedly-optimal $X''$, giving a contradiction.

Another example

\[
\begin{array}{c|cccccc}
A & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 \\
\hline
s(A) & 1 & 1 & 5 & 8 & 7 & 11 & 3 \\
f(A) & 2 & 4 & 6 & 9 & 10 & 12 & 13 \\
\end{array}
\]

(to speed things up activities are already sorted by non-decreasing finish time).
A related problem

**Weighted Activity-selection problem.**

Suppose that each activity $A \in S$ has a positive integer value $w(A)$.

The *weighted* activity-selection problem is to select a number of mutually compatible activities of the largest possible total weight.

You can think of $w(A)$ as the profit gained from doing activity $A$.

How would you solve the problem?

An example...

One more example

... and again
No (optimal) greedy algorithm is known for the Weighted Activity Selection problem.

The Greedy Choice property seems to fail! An activity that ends earliest is not guaranteed to be included in an optimal solution.

Note that proving that a greedy algorithm does not work is much easier than proving that it does work — you just need to provide an input on which it is not optimal.

Digression: Let’s look at another example where a greedy algorithm does not work.

How do we solve weighted activity selection?

Divide and Conquer

Let \( S = \{ A_1, \ldots, A_n \} \). Assume that the activities are sorted by nondecreasing finish time so \( f(A_1) \leq f(A_2) \leq \cdots \leq f(A_n) \).

For each \( j \), define \( p(j) \) to be the largest index smaller than \( j \) of an activity compatible with \( A_j \) (\( p(j) = 0 \) if such index does not exist).

Let’s look at the \( p(j) \) values for an example.

In CS202, you considered the Fractional Knapsack Problem.
The input was a set of \( n \) items, in which the \( i \)th item has benefit \( b_i \geq 0 \) and weight \( w_i \geq 0 \). Also a weight limit \( W \).

In the output, we choose a portion \( x_i \) of item \( i \) \( (0 \leq x_i \leq w_i) \) so total weight is \( \sum x_i \leq W \). Maximise \( \sum b_i x_i / w_i \).

The greedy solution that you studied picks items in order of decreasing \( b_i / w_i \). This gives the optimal solution.

What if we insist \( x_i \in \{0, w_i\} \)?

\( b_1 = 60, w_1 = 10, b_2 = 100, w_2 = 20, b_3 = 120, w_3 = 30, W = 50 \)

Greedy: items 1 and 2. Better: items 2 and 3.
Case 1. \( A_n \) in \( X \): then \( X \) cannot contain any other activity with index larger than \( p(n) \).

Case 2. \( A_n \) is not in \( X \). We can throw it away and consider \( A_1, \ldots, A_{n-1} \).

To summarize either the optimal solution is formed by some solution to the instance formed by activities \( A_1, \ldots, A_{p(n)} \) plus \( A_n \) or the optimal solution does not contain \( A_n \) (and therefore it is some solution to the instance formed by activities \( A_1, \ldots, A_{n-1} \)).

So, to to find the solution to the whole instance, we should choose between the optimal solutions of these two subproblems.

In other words

\[
\text{MaxWeight}(n) = \max \{ w(A_n) + \text{MaxWeight}(p(n)), \text{MaxWeight}(n-1) \}.
\]

Recursive solution

Based on the argument so far we can at least write a recursive method that computes the weight of an optimal solution ... rather inefficiently.

```
RECURSIVE-WEIGHTED-SELECTOR (j)
if \( j = 0 \)
    return 0
else
    return \( \max \{ w(A_j) + \text{RECURSIVE-WEIGHTED-SELECTOR (p(j))}, \text{RECURSIVE-WEIGHTED-SELECTOR (j - 1)} \} \)
```

The optimum for the given instance on \( n \) activities is obtained by running \text{RECURSIVE-WEIGHTED-SELECTOR (n)}. 
Memoization

The array $M[j]$ (another global variable) contains the size of the optima for the instances $A_1, \ldots, A_j$.

\begin{verbatim}
FAST-REC-WEIGHTED-SELECTOR (j)
  if j = 0
    return 0
  else if M[j] not empty
    return M[j]
  else
    M[j] ← max (w(A_j) + FAST-REC-WEIGHTED-SELECTOR (p(j)),
                  FAST-REC-WEIGHTED-SELECTOR (j − 1))
  return M[j]
\end{verbatim}

Exercises

1. Derive a recursive algorithm that actually computes the optimal set of activities from the algorithm RECURSIVE-WEIGHTED-SELECTOR.
2. Derive a recursive algorithm that actually computes the optimal set of activities from the algorithm FAST-REC-WEIGHTED-SELECTOR.
3. Simulate both RECURSIVE-WEIGHTED-SELECTOR and FAST-REC-WEIGHTED-SELECTOR on the following instance:

\begin{verbatim}
<table>
<thead>
<tr>
<th>A</th>
<th>A_1</th>
<th>A_2</th>
<th>A_3</th>
<th>A_4</th>
<th>A_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>s(A)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>f(A)</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>w(A)</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
\end{verbatim}

Dynamic Programming

Dynamic programming, like the divide-and-conquer method used in quicksort, solves problems by combining the solutions to subproblems.

Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
If two subproblems share subsubproblems then a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems.

In contrast, dynamic programming is applicable (and often advisable) when the subproblems are not independent, that is, when subproblems share subsubproblems. A dynamic programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered.

Dynamic programming is typically applied to optimisation problems.

The development of a dynamic programming algorithm can be broken into a sequence of steps.

1. Show how you can get an optimal solution by combining optimal solutions to subproblems.
2. Show how you can recursively compute the value of an optimal solution using the values of optimal solutions to subproblems.
3. Compute the value of an optimal solution bottom-up by first solving the simplest subproblems.
4. Save enough information to allow you to construct an optimal solution (and not merely its value).

Example: matrix-chain multiplication

Background: multiplying two matrices

\[
\text{cols}(A) = \text{rows}(B) \quad \text{cols}(B) = \text{cols}(C)
\]

\[
\begin{align*}
\text{rows}(A) \times \text{columns}(B) \times \text{columns}(A)
\end{align*}
\]

Number of scalar multiplications:

\[
\text{rows}(A) \times \text{columns}(B) \times \text{columns}(A)
\]
Matrix-Multiply \((A, B)\)

\[
\begin{align*}
\text{For } i & \leftarrow 1 \text{ to } \text{rows}(A) \\
\text{For } j & \leftarrow 1 \text{ to } \text{columns}(B) \\
C[i,j] & \leftarrow 0 \\
\text{For } k & \leftarrow 1 \text{ to } \text{columns}(A) \\
C[i,j] & \leftarrow C[i,j] + A[i,k] \cdot B[k,j]
\end{align*}
\]

Given a chain of matrices to multiply, any parenthesization is valid (dimensions are OK, so multiplication makes sense)

Given a chain of matrices to multiply, any parenthesization gives the same answer (matrix multiplication is associative)

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3
\end{bmatrix}
\times
\begin{bmatrix}
1 & 2 \\
2 & 4 \\
3 & 6 \\
4 & 8 \\
5 & 10
\end{bmatrix}
\]

\[
\begin{bmatrix}
15 & 30 \\
30 & 60 \\
45 & 90
\end{bmatrix}
\times
\begin{bmatrix}
-1 & 1 \\
-1 & 1
\end{bmatrix}
= 
\begin{bmatrix}
15 & 30 \\
30 & 60 \\
45 & 90
\end{bmatrix}, \text{ or}
\]

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3
\end{bmatrix}
\times
\begin{bmatrix}
1 & 2 \\
2 & 4 \\
3 & 6 \\
4 & 8 \\
5 & 10
\end{bmatrix}
= 
\begin{bmatrix}
15 & 30 \\
30 & 60 \\
45 & 90
\end{bmatrix}
\]

To see that you get the same answer, look at the top left corner of the output...

\[
\begin{bmatrix}
a_1 & a_2 & a_3 \\
\vdots & \vdots & \vdots \\
& b_2 & b_5 \\
\end{bmatrix}
\times
\begin{bmatrix}
x_1 & \cdots \\
x_2 & \cdots
\end{bmatrix}
= 
\begin{bmatrix}
a_1(x_1b_1 + x_2b_4) + a_2(x_1b_2 + x_2b_5) + a_3(x_1b_3 + x_2b_6)
\end{bmatrix}
\]

\[
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
200 & 50 \\
50 & 85
\end{bmatrix}
\]

\[
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
200 & 50 \\
50 & 85
\end{bmatrix}
\]

or

\[
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
200 & 50 \\
50 & 85
\end{bmatrix}
\]

\[
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
200 & 50 \\
50 & 85
\end{bmatrix}
\]

or

\[
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
100 & 200 \\
50 & 85
\end{bmatrix}
\times
\begin{bmatrix}
200 & 50 \\
50 & 85
\end{bmatrix}
\]
However, different parenthesizations lead to different numbers of scalar multiplications. Consider this example.

Method 1: Multiplying $M_3$ by $M_4$ costs 200,000 scalar multiplications.

Then multiplying $M_1$ by $M_2$ costs 800,000 scalar multiplications.

Method 2: Multiplying $M_2$ by $M_3$ costs 40,000 scalar multiplications.

Then multiplying $M_1$ by $M_2 \cdot M_3$ costs 88,000 scalar multiplications. The total number of scalar multiplications is only 88,000 (as opposed to 1,160,000 when we chose the parentheses according to method 1).
Problem definition

Given a sequence of matrices $A_1, \ldots, A_N$ such that $A_i$ has dimensions $n_i \times n_{i+1}$, find the parenthesization of $A_1 \times \ldots \times A_N$ that minimises the number of scalar multiplications (assuming that each matrix multiplication is done using Matrix-Multiply).

This is an optimisation problem. Each instance is a sequence of integers $n_1, n_2, \ldots, n_{N+1}$. A solution to the instance is an ordering of the $N - 1$ multiplications. The cost of an ordering is the number of scalar multiplications performed and the problem seeks a minimum-cost ordering.

Note that $y_1$ is an optimal solution for the instance $A_{1..k}$ (otherwise the supposedly-optimal solution $y$ can be improved!). Similarly, $y_2$ is an optimal solution for the instance $A_{k+1..N}$.

Thus, an optimal solution to an instance of the matrix-chain multiplication problem contains within it optimal solutions to subproblem instances!

Next we present a dynamic programming algorithm for solving this problem. We follow the method given earlier.

Getting an optimal solution out of optimal solutions to subproblems

Denote $A_i \times \ldots \times A_j$ by $A_{i..j}$.

An optimal ordering $y$ of $A_{1..N}$ splits the product at some matrix $A_k$: $A_1 \times \ldots \times A_N = A_{1..k} \times A_{k+1..N}$

Then if $y_1$ denotes the ordering for $A_{1..k}$ in $y$ and $y_2$ denotes the ordering for $A_{k+1..N}$ in $y$ we have

$$
cost(A_{1..N}, y) = cost(A_{1..k}, y_1) + cost(A_{k+1..N}, y_2) + \text{mult}(A_{1..k}, A_{k+1..N})
$$

where $\text{mult}(A, B)$ denotes the number of scalar multiplications performed in a call to Matrix-Multiply$(A, B)$.

How to recursively compute the value of an optimal solution using the values of optimal solutions to subproblems

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$.

The cost of a cheapest way to compute $A_{1..N}$ is $m[1, N]$. 
Definition of \(m[i,j]\)

- If \(i = j\), the chain consists of a single matrix \(A_{i,i} = A_i\), no scalar multiplication is needed and therefore \(m[i,i] = 0\).
- If \(i < j\) then \(m[i,j]\) can be computed by taking advantage of the structure of an optimal solution, as described earlier. Therefore, if the optimal cost ordering of \(A_{i..j}\) is obtained multiplying \(A_{i..k}\) and \(A_{k+1..j}\) then we can define

\[
m[i,j] = m[i,k] + m[k+1,j] + \text{mult}(A_{i..k}, A_{k+1..j}).
\]

Notice that \(A_{i..k}\) is a \(n_i \times n_{k+1}\) matrix and \(A_{k+1..j}\) is an \(n_{k+1} \times n_{j+1}\) matrix. Therefore multiplying them takes \(n_{k+1} \cdot n_i \cdot n_{j+1}\) multiplications. Hence

\[
m[i,j] = m[i,k] + m[k+1,j] + n_{k+1} \cdot n_i \cdot n_{j+1}.
\]

... but, alas! We do not know \(k\)!!!

Computing the optimal costs

Here is a simple recursive program that, given the sequence \(n_1, n_2, \ldots, n_{N+1}\), computes \(m[1,N]\).

```plaintext
Matrix-Chain (i,j)
If i=j
  Return 0
Else
  C ← ∞
  For k ← i to j-1
    Z ← Matrix-Chain(i,k) + Matrix-Chain(k+1,j) + n_{k+1}n_in_{j+1}
    If Z < C
      C ← Z
  Return C
```

No problem, there can only be \(j-i\) possible values for \(k\), we can try all of them. Thus, we have

\[
m[i,j] = \begin{cases} 0 & i = j \\ \min_{i≤k<j} \{m[i,k] + m[k+1,j] + n_{k+1}n_{j+1}\} & i < j \end{cases}
\]

How long does the recursive algorithm take? Let \(n = j-i\).

\[
T(n) = n + \sum_{\ell=1}^{n-1} T(\ell) + T(n-\ell)
= n + 2 \sum_{\ell=1}^{n-1} T(\ell)
\]

Prove by induction on \(n\) that \(T(n) \geq 2^{n-1}\)

Base case. \(n = 1\). Inductive step.

\[
T(n) \geq n + 2 \sum_{\ell=0}^{n-2} 2^\ell
= n + 2(2^{n-1} - 1)
\geq 2^{n-1}
\]
Key Observation!

There are relatively few subproblems: one for each choice of \( i \) and \( j \) (that's \( \binom{N}{2} + N \) in total, \( \binom{N}{2} \) for pairs with \( i < j \) and \( N \) more for pairs with \( i = j \)).

Instead of computing the solution to the recurrence recursively (top-down), we perform the third step of the dynamic programming paradigm and compute the optimal costs of the subproblems using a bottom-up approach.

More intuition

Now, focus on

\[
m[i, j] = \min_k m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \quad \text{for} \quad i \leq k < j.
\]

The figure on the right shows how the entries of \( m[i, j] \) are filled. For each fixed value of \( \ell \) the process fills the entries that are \( \ell \) places to the right of the main diagonal. Notice that all that is needed to compute \( m[i, j] \) are the entries \( m[i, k] \) and \( m[k + 1, j] \) for \( i \leq k < j \).

For example, \( m[1, 5] \) will need \( m[1, 1] \) and \( m[2, 5] \), or \( m[1, 2] \) and \( m[3, 5] \), and so on.

Example instance: \( N = 6 \) and

\[
m[i, j] = \min_k m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \quad \text{for} \quad i \leq k < j.
\]

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
i \backslash j & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 12 & 5 & 2 & 5 & 8 & 4 \\
2 & 0 & 2*5 & 0 & 0 & 0 & 0 \\
3 & 0 & 2*8 & 0 & 0 & 0 & 0 \\
4 & 0 & 5*4 & 0 & 0 & 0 & 0 \\
5 & 0 & 8*7 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

for \( \ell = 1 \) we have \( j = i + \ell \) and the only choice is \( k = i \) (which gives \( k + 1 = j \)).
Example instance: $N = 6$ and $i_n = 12 5 2 5 4 7$

$$m[i, j] = \min_k m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \text{ for } i \leq k < j.$$ 

<table>
<thead>
<tr>
<th>$i \setminus j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>120</td>
<td>240</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>50</td>
<td>160</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>80</td>
<td>144</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>160</td>
<td>300</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for $\ell = 2$ we have $j = i + \ell$ and the choices are $k = i$ and $k = i + 1$. For example, for $i = 1$ and $j = 3$, we have two choices.

$k = 1 : 0 + 50 + 12 \cdot 5 \cdot 5 = 350$

$k = 2 : 120 + 0 + 12 \cdot 2 \cdot 5 = 240$

Example instance: $N = 6$ and $i_n = 12 5 2 5 4 7$

$$m[i, j] = \min_k m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \text{ for } i \leq k < j.$$ 

<table>
<thead>
<tr>
<th>$i \setminus j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>120</td>
<td>240</td>
<td>392</td>
<td>360</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>50</td>
<td>160</td>
<td>184</td>
<td>270</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>80</td>
<td>144</td>
<td>200</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>160</td>
<td>300</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for $\ell = 3$ we have $j = i + \ell$ and the choices are $k = i$, $k = i + 1$, and $k = i + 2$. For example, for $i = 2$ and $j = 5$, we have these choices.

$k = 2 : 0 + 144 + 5 \cdot 2 \cdot 4 = 184$

$k = 3 : 50 + 160 + 5 \cdot 5 \cdot 4 = 310$

$k = 4 : 160 + 0 + 5 \cdot 8 \cdot 4 = 320$

Example instance: $N = 6$ and $i_n = 12 5 2 5 4 7$

$$m[i, j] = \min_k m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \text{ for } i \leq k < j.$$ 

<table>
<thead>
<tr>
<th>$i \setminus j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>120</td>
<td>240</td>
<td>392</td>
<td>360</td>
<td>488</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>50</td>
<td>160</td>
<td>184</td>
<td>270</td>
<td>270</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>80</td>
<td>144</td>
<td>200</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>160</td>
<td>300</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>224</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

488 is the answer - this is the number of scalar multiplications needed to multiply $A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6$. 
What is the complexity of the algorithm?

\[
\text{\textsc{Matrix-Chain-Order}} \ (n_1, n_2, \ldots, n_{N+1}, N) \\
\text{// First, fill all the elements in the diagonal with zero} \\
\text{for } i \leftarrow 1 \text{ to } N \\
\quad m[i, i] \leftarrow 0 \\
\text{// Next, fill all elements at distance } \ell \text{ from the diagonal} \\
\text{for } \ell \leftarrow 1 \text{ to } N - 1 \\
\quad \text{for } i \leftarrow 1 \text{ to } N - \ell \\
\quad \quad j \leftarrow i + \ell \\
\quad \quad \text{define } m[i, j] \text{ as the minimum of} \\
\quad \quad \quad m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \\
\quad \quad \text{for } i \leq k < j.
\]

\[O(N^3)\]

Comparing Running times

Michele’s Java implementation of both algorithms, run on a Linux 200MHz PC on \(N\) square matrices of size 3.

<table>
<thead>
<tr>
<th>(N)</th>
<th>time</th>
<th>scalar mults</th>
<th>(N)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1.20</td>
<td>135</td>
<td>6</td>
<td>1.05</td>
</tr>
<tr>
<td>8</td>
<td>1.04</td>
<td>189</td>
<td>8</td>
<td>1.04</td>
</tr>
<tr>
<td>10</td>
<td>1.10</td>
<td>243</td>
<td>10</td>
<td>1.09</td>
</tr>
<tr>
<td>12</td>
<td>1.15</td>
<td>297</td>
<td>12</td>
<td>1.15</td>
</tr>
<tr>
<td>14</td>
<td>1.78</td>
<td>351</td>
<td>14</td>
<td>1.08</td>
</tr>
<tr>
<td>16</td>
<td>6.86</td>
<td>405</td>
<td>16</td>
<td>1.11</td>
</tr>
<tr>
<td>18</td>
<td>51.40</td>
<td>459</td>
<td>18</td>
<td>1.12</td>
</tr>
<tr>
<td>20</td>
<td>7:32.80</td>
<td>513</td>
<td>20</td>
<td>1.07</td>
</tr>
<tr>
<td>22</td>
<td>1:11:20.64</td>
<td>567</td>
<td>22</td>
<td>1.09</td>
</tr>
</tbody>
</table>

Constructing an optimal solution

We can use another table \(s\) to allow us to compute the optimal parenthesization. Each entry \(s[i, j]\) records the value \(k\) such that the optimal ordering to compute \(A_i \times \ldots \times A_j\) splits the product as

\[(A_i \times \ldots \times A_k)(A_{k+1} \times \ldots \times A_j).\]

Thus we know that the final ordering for computing \(A_1 \ldots N\) optimally is \(A_1 \ldots s[1, N] \times A_{s[1, N]+1} \ldots N\). Earlier matrix multiplications can be computed recursively.

\[
\text{\textsc{Matrix-Chain-Order}} \ (n_1, n_2, \ldots, n_{N+1}, N) \\
\text{for } i \leftarrow 1 \text{ to } N \\
\quad m[i, i] \leftarrow 0 \\
\text{for } \ell \leftarrow 1 \text{ to } N - 1 \\
\quad \text{for } i \leftarrow 1 \text{ to } N - \ell \\
\quad \quad j \leftarrow i + \ell \\
\quad \quad m[i, j] \leftarrow +\infty \\
\quad \quad \text{for } k \leftarrow i \text{ to } j - 1 \\
\quad \quad \quad q \leftarrow m[i, k] + m[k + 1, j] + n_i n_{k+1} n_{j+1} \\
\quad \quad \quad \text{if } q < m[i, j] \\
\quad \quad \quad \quad m[i, j] \leftarrow q \\
\quad \quad \quad s[i, j] \leftarrow k
\]
Another example of dynamic programming: The All-Pairs Shortest Paths Problem

**Input:** An $n \times n$ matrix $W$ in which $W[i, j]$ is the weight of the edge from $i$ to $j$ in a graph. $W$ must satisfy the following.

- $W[i, i] = 0$
- $W[i, j] = \infty$ if there is no edge from $i$ to $j$.
- There are no negative-weight cycles in the weighted digraph corresponding to $W$.

**Output:** An $n \times n$ matrix $D$ in which $D[i, j]$ is the weight of a shortest (smallest-weight) path from $i$ to $j$.

For example, consider the undirected graph with vertices \{1, 2, 3, 4\} and $W[1, 2] = 2$, $W[2, 3] = 2$, and $W[1, 3] = 4$.

A dynamic-programming algorithm

```plaintext
for i ← 1 to n
    for j ← 1 to n
        D[0, i, j] ← ∞
for i ← 1 to n
    D[0, i, i] ← 0
for m ← 1 to n - 1
    for i ← 1 to n
        for j ← 1 to n
            D[m, i, j] ← D[m - 1, i, 1] + W[1, j]
            for k ← 2 to n
                If $D[m, i, j] > D[m - 1, i, k] + W[k, j]$
                $D[m, i, j] ← D[m - 1, i, k] + W[k, j]$
```

Let $D[m, i, j]$ be the minimum weight of any path from $i$ to $j$ that contains at most $m$ edges.

$$D[m, i, j] = \min \{D[m - 1, i, k] + W[k, j] \mid 1 \leq k \leq n \}.$$
Simulating the algorithm

\[ D[0, i, j] = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & \infty & \infty & \infty & \infty \\
2 & \infty & 0 & \infty & \infty \\
3 & \infty & \infty & 0 & \infty \\
4 & \infty & \infty & \infty & 0 \\
5 & \infty & \infty & \infty & 0 \\
\end{bmatrix} \]

\[ D[1, i, j] = \min \{ D[0, i, k] + W[k, j] \mid 1 \leq k \leq n \} = W[i, j]. \]

\[ D[1, i, j] = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 3 & 8 & \infty & -4 \\
2 & \infty & 0 & \infty & 1 & 7 \\
3 & \infty & 4 & 0 & \infty & \infty \\
4 & 2 & \infty & -5 & 0 & \infty \\
5 & \infty & \infty & 6 & 0 \\
\end{bmatrix} \]

\[ D[2, i, j] = \min \{ D[1, i, k] + W[k, j] \mid 1 \leq k \leq n \}. \]

\[ D[2, i, j] = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 3 & 8 & 2 & -4 \\
2 & \infty & 0 & \infty & -4 & 1 & 7 \\
3 & \infty & 4 & 0 & 5 & 11 \\
4 & 2 & \infty & -5 & 0 & -2 & \infty \\
5 & 8 & \infty & 1 & 6 & 0 \\
\end{bmatrix} \]

\[ D[3, i, j] = \min \{ D[2, i, k] + W[k, j] \mid 1 \leq k \leq n \}. \]

\[ D[3, i, j] = \begin{bmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 3 & -3 & 2 & -4 \\
2 & 3 & 0 & -4 & 1 & 7 \\
3 & 7 & 4 & 0 & 5 & 11 \\
4 & 2 & -1 & -5 & 0 & -2 \\
5 & 8 & 5 & 1 & 6 & 0 \\
\end{bmatrix} \]
$$D[3, i, j] = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & -3 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & -1 \\ 3 & 7 & 4 & 0 & 5 & 11 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

$$W[i, j] = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \end{bmatrix}$$

$$D[4, i, j] = \min\{ D[3, i, k] + W[k, j] \mid 1 \leq k \leq n \}.$$ 

$$D[4, i, j] = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & -3 & 2 & -4 \\ 2 & 3 & 0 & -4 & 1 & -1 \\ 3 & 7 & 4 & 0 & 5 & 3 \\ 4 & 2 & -1 & -5 & 0 & -2 \\ 5 & 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

The time complexity is $O(n^4)$. There are faster dynamic-programming algorithms.

**Exercise**

**Golf playing.** A target value $N$ is given, along with a set $S = \{s_1, s_2, \ldots, s_n\}$ of admissible segment lengths and a set $B = \{b_1, \ldots, b_m\}$ of forbidden values. The aim is to choose the shortest sequence $\sigma_1, \sigma_2, \ldots, \sigma_u$ of elements of $S$ such that

1. $\sum_{i=1}^{u} \sigma_i = N$ and
2. $\sum_{i=1}^{j} \sigma_i \notin B$ for each $j \in \{1, \ldots, u\}$.

**Claim.** Any instance of this problem is solvable optimally in time polynomial in $N$ and $n$.

**Exercise**

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of $n$ words of length $\ell_1, \ldots, \ell_n$ (number of characters). We want to print this paragraph neatly on a number of lines that hold a maximum of $M$ characters each. Our criterion of “neatness” is as follows. If a given line contains words $i$ through $j$, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is

$$M - j + i - \sum_{k=i}^{j} \ell_k,$$

(which must be non-negative so that the word fit on the line). We wish to minimise the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines.

**Dynamic programming solution?**

**Run-time? Space requirements?**