Dynamic Programming

- We have previously looked at two different types of algorithms
  - **Greedy** algorithms where the choice made at each step is simply what looks best at that moment. That is it makes a *locally optimal* choice.
  - Examples include the fractional knapsack problem, Huffman coding and Dijkstra’s shortest path algorithm
  - **Divide and Conquer** algorithms where the problem is split into parts which are solved and the results then merged to solve the whole problem
  - Examples include Merge Sort, Karatsuba multiplication. Order statistics
- For many problems we need a different approach called **Dynamic Programming**
In this lecture, we look at **dynamic programming** methods.

This is one of the key *sledgehammers* in algorithms along with linear programming, and the lighter hammers of divide and conquer and simple greedy approaches:

- Finds efficient solutions for problems with lots of overlapping sub-problems.
- Essentially, we try to solve each sub-problem only once.

The name is a little misleading, but comes historically from:

- Richard Bellman, a professor of mathematics at Stanford University around 1950.
- In those days programming had a meaning closer to planning.

The main ideas behind dynamic programming that we will discuss are:

- Overlapping subproblems
- Optimal substructure
- Memoization
Typically, a dynamic programming solution is constructed using a series of steps.

- Characterise the *structure* of an optimal solution.
- *Recursively* define the value of an optimal solution.
- Compute the value of an optimal solution in a **bottom-up** or **top-down** fashion.
  - That is, build it from the results of smaller solutions either iteratively from the bottom or recursively from the top.
- Construct the final solution from the computed information.

We can omit the last step if only the value, rather than the method to calculate it, is required.
A Simple Example (1)

- As a very simplistic example of dynamic programming, consider calculating the **Fibonacci sequence**:
  - The $n$-th number is the sum of the previous two.
- This can be implemented using a simple recursive algorithm:
  
  ```
  FIBONACCI(n) 
  begin 
    if $n = 0$ then 
      return 0 
    if $n = 1$ then 
      return 1 
    return FIBONACCI($n - 1$) + FIBONACCI($n - 2$) 
  end 
  ```

- Note how the computation involved in the algorithm relates to our definition of dynamic programming:
  - There are overlapping sub-problems, for example computing $FIBONACCI(n - 1)$ overlaps $FIBONACCI(n - 2)$. 

A Simple Example (2)

Unfortunately the recursive formulation as given in the previous slide takes exponential time!

- It recomputes the same values of FIBONACCI over and over again
- We need to memoize
- Haskell is particularly good at this if you use the full power of infinite lists and lazy evaluation

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
fastfibs n = fibs !! n
```

- zipWith is a function defined in the Haskell prelude which builds a new list by taking the head element of each list and applying a function to them

```haskell
zipWith (+) [0,1,2] [3,4,5] [3,5,7]
```
A Simple Example (3)

- To apply dynamic programming in an imperative programming language, we can make an array $f$ and compute the $n$-th Fibonacci number in a bottom-up way as follows:

  \begin{align*}
  \text{Pre-computation} \\
  f[0] &\leftarrow 0 \\
  f[1] &\leftarrow 1
  \end{align*}

  \begin{align*}
  \text{Main Algorithm} \\
  \text{FIBONACCI}(n) \\
  \text{begin} \\
  \quad \text{for } i = 2 \text{ upto } n \text{ step } 1 \text{ do} \\
  \quad \quad f[i] &\leftarrow f[i - 1] + f[i - 2] \\
  \quad \text{return } f[n] \\
  \text{end}
  \end{align*}

- We have converted a recursive formulation into an iterative one

  - We could have made an efficient recursive formulation by explicitly storing the results of each calculation in a global array
  - We would then have had to check if the result was already known at each recursive call
A more sophisticated example of dynamic programming involves matrix multiplication chains.

We are given a sequence or chain of \( n \) matrices \(< A_1, A_2, \ldots, A_n >\).

The goal is to compute \( A_1 \cdot A_2 \cdot \cdots \cdot A_n \), the product of all the matrices.

Note that not all combinations of \( A_i \cdot A_j \) are valid:

- Two matrices \( A_i \) and \( A_j \) are **compatible** if the number of columns in \( A_i \) equals the number of rows in \( A_j \).
- We can only multiply together matrices that are compatible.
Avoiding compatibility for a moment, we need to decide which order to do the multiplications in:

\[
(A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))) \\
(A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)) \\
((A_1 \cdot A_2) \cdot (A_3 \cdot A_4)) \\
((A_1 \cdot (A_2 \cdot A_3)) \cdot A_4) \\
(((A_1 \cdot A_2) \cdot A_3) \cdot A_4)
\]

Since matrix multiplication is \textit{associative}, all these different orders are equivalent in terms of the result.

The question then is why would we ever want to use one multiplication order over another?
Matrix-chain Multiplication (3)

- One reason is to do with the number of elemental multiplications we need to do: that is, the number of $A_1[x_1, y_1] \cdot A_2[x_2, y_2]$ type multiplications.

- Say we have the chain $< A_1, A_2, A_3 >$ where sizes of $A_1$, $A_2$ and $A_3$ are $10 \times 100$, $100 \times 5$ and $5 \times 50$.

- Using the order $((A_1 \cdot A_2) \cdot A_3)$ uses:
  \[ (10 \cdot 100 \cdot 5) + (10 \cdot 5 \cdot 50) = 7500 \text{ multiplications} \]

- Using the order $(A_1 \cdot (A_2 \cdot A_3))$ uses:
  \[ (10 \cdot 100 \cdot 50) + (100 \cdot 5 \cdot 50) = 75000 \text{ multiplications} \]

- Using the first order is clearly more attractive than the second!
Matrix-chain Multiplication (4)

- A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?
  - Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$
  - When $n \geq 2$, $P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \cdots + P(n - 1)P(1)$
  - Guess the solution is $\Omega(2^n)$ and use the subsitution method to check.
  - The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$
  - Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n - k) \geq \sum_{k=1}^{n-1} c2^k2^{n-k}$ for some constant $c$ and all $n > n_0$
  - Therefore $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n - 1)2^n \geq c2^n$
  - By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!
  - We really do need to find a faster solution
A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?

Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$

When $n \geq 2$, $P(n) = P(1)P(n-1) + P(2)P(n-2) + \cdots + P(n-1)P(1)$

Guess the solution is $\Omega(2^n)$ and use the substitution method to check.

The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$

Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} c2^k2^{n-k}$ for some constant $c$ and all $n > n_0$

Therefore $P(n) \geq c\sum_{k=1}^{n-1} 2^n = c(n-1)2^n \geq c2^n$

By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!

We really do need to find a faster solution.
A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?

Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$.

When $n \geq 2$, $P(n) = P(1)P(n-1) + P(2)P(n-2) + \cdots + P(n-1)P(1)$.

Guess the solution is $\Omega(2^n)$ and use the substitution method to check.

The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$.

Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} c2^k 2^{n-k}$ for some constant $c$ and all $n > n_0$.

Therefore $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n-1)2^n \geq c2^n$

By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!

We really do need to find a faster solution.
A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?

Call the number of possible bracketings \( P(n) \). When \( n = 1 \) there is just one matrix, therefore \( P(1) = 1 \).

When \( n \geq 2 \), \( P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \cdots + P(n - 1)P(1) \).

Guess the solution is \( \Omega(2^n) \) and use the substitution method to check.

The inductive hypothesis is that \( P(k) \in \Omega(2^k) \) for all \( 1 < k < n \).

Therefore \( P(n) = \sum_{k=1}^{n-1} P(k)P(n - k) \geq \sum_{k=1}^{n-1} c2^k2^{n-k} \) for some constant \( c \) and all \( n > n_0 \).

Therefore \( P(n) \geq c \sum_{k=1}^{n-1} 2^k = c(n - 1)2^n \geq c2^n \).

By induction we have shown that the number of bracketings grows at least as fast as \( 2^n \). This is really bad!

We really do need to find a faster solution.
A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?

- Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$
- When $n \geq 2$, $P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \cdots + P(n - 1)P(1)$
- Guess the solution is $\Omega(2^n)$ and use the substitution method to check.
- The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$
- Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n - k) \geq \sum_{k=1}^{n-1} c2^k 2^{n-k}$ for some constant $c$ and all $n > n_0$
- Therefore $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n - 1)2^n \geq c2^n$
- By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!
- We really do need to find a faster solution
Matrix-chain Multiplication (4)

- A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?
  - Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$
  - When $n \geq 2$, $P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \cdots + P(n - 1)P(1)$
  - Guess the solution is $\Omega(2^n)$ and use the substitution method to check.
  - The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$
  - Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n - k) \geq \sum_{k=1}^{n-1} c2^k2^{n-k}$ for some constant $c$ and all $n > n_0$
  - Therefore $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n - 1)2^n \geq c2^n$
  - By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!
  - We really do need to find a faster solution
Matrix-chain Multiplication (4)

- A naive technique to solve this problem might be to enumerate all the different possible bracketings and select the one with the minimal cost. How many possible bracketings are there?
  - Call the number of possible bracketings $P(n)$. When $n = 1$ there is just one matrix, therefore $P(1) = 1$
  - When $n \geq 2$, $P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \cdots + P(n - 1)P(1)$
  - Guess the solution is $\Omega(2^n)$ and use the substitution method to check.
  - The inductive hypothesis is that $P(k) \in \Omega(2^k)$ for all $1 < k < n$
  - Therefore $P(n) = \sum_{k=1}^{n-1} P(k)P(n - k) \geq \sum_{k=1}^{n-1} c2^k2^{n-k}$ for some constant $c$ and all $n > n_0$
  - Therefore $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n - 1)2^n \geq c2^n$
  - By induction we have shown that the number of bracketings grows at least as fast as $2^n$. This is really bad!
  - We really do need to find a faster solution
The first task is to **characterise the structure** of an optimal ordering.

Say we have a chain of matrices $< A_i, A_{i+1}, \ldots, A_{j-1}, A_j >$. We denote the product of these matrices by $A_{i\ldots j}$.

To compute the result, we need to do the following:

- Pick some $k$ where $i \leq k < j$ that splits the chain into two parts.
- Calculate the product of the two parts $A_{i\ldots k}$ and $A_{k+1\ldots j}$.
- Multiply them together to get the final result.

Clearly each part must also be optimally ordered, otherwise the entire solution isn’t optimal!

The problem then is to select where to **split** the chain.
The next task is to formulate a method for computing solutions.

Given the same chain \(<A_i, A_{i+1}, \ldots, A_{j-1}, A_j>\) we let \(m[i, j]\) denote the cost of computing the result \(A_{i\ldots j}\):

- Some of the entries are trivial, for example \(m[i, i]\).
- Most are harder: if we split the chain at \(k\) again, the cost of \(m[i, j]\) is equal to the cost of computing the two parts plus multiplying them together.

Hence, we can formalise the optimal cost as follows:

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j \\
\min_k (m[i, k] + m[k + 1, j] + c) & \text{if } i \neq j 
\end{cases}
\]

For now, we assume some constant \(c\) that denotes the cost of multiplying the two parts together.
The next problem is how we actually compute an \( m[i, j] \) result for given \( i \) and \( j \).

The problem is, if we write a recursive algorithm the overlapping sub-problems bite us:

- In general, a recursive solution will have exponential complexity which is bad news.
- However, such a solution will encounter common sub-problems in different parts of the recursion tree.
- So we can again use this fact to reduce the complexity to a more manageable form.

The plan is, instead of working recursively we calculate in a bottom-up way and reuse common results.
First some more notation to make the algorithm a bit clearer:

- To keep track of the optimal solution and avoid having to recover it at the end, we let $s[i, j]$ equal the value of $k$ that minimises $m[i, k] + m[k + 1, j] + c$.
- We let the matrix $A_i$ have dimension $p_{i-1} \times p_i$ so that the cost $c$ is actually $p_{i-1} \cdot p_k \cdot p_j$.

And some details of the algorithm:

- As input, it takes a list of dimensions $p = (p_0, p_1, \ldots, p_n)$ so that the length of $p$ is $n + 1$.
- It uses two tables $m[1 \ldots n, 1 \ldots n]$ and $s[1 \ldots n, 1 \ldots n]$ to hold the intermediate results.
- The $s$ table is eventually used to compute an optimal solution which is the output.
Matrix-chain Multiplication (9)

► The final algorithm reads as follows:

```plaintext
CALCULATE-ORDER(p)
begin
  n ← |p| − 1
  for i = 1 upto n step 1 do
    m[i, i] ← 0
  for l = 2 upto n step 1 do
    for i = 1 upto n − l + 1 step 1 do
      j ← i + l − 1
      m[i, j] ← ∞
      for k = i upto j − 1 step 1 do
        q ← m[i, k] + m[k + 1, j] + (p_i−1 ∙ p_k ∙ p_j)
        if q < m[i, j] then
          m[i, j] ← q
          s[i, j] ← k
      return (m, s)
end
```

► It doesn’t take too much effort to see that it is $O(n^3)$. 
Matrix-chain Multiplication (10)

- The first loop calculates all the trivial entries.
- The rest calculates the non-trivial entries using our cost formula:
  - At each step we only depend on previous results already calculated, this is the really important bit.
- The operation is best illustrated with an example. Consider the case where we have the following matrices:

  \[
  A_1: 30 \times 35 \quad A_2: 35 \times 15 \\
  A_3: 15 \times 5 \quad A_4: 5 \times 10 \\
  A_5: 10 \times 20 \quad A_6: 20 \times 25 
  \]

- Note that from this input \( p = (30, 35, 15, 5, 10, 20, 25) \).
Matrix-chain Multiplication (11)

From this list of matrices, we produce $m$ and $s$ tables:

Note that we only consider $i \leq j$ so only the upper part of the table is valid, that is why the tables are a funny shape.
The slightly confusing bit is how the $i$ and $j$ loops work:

- Basically they work from left to right, bottom to top across the pyramid shape of the $m$ and $s$ tables.

Given an $i$ and $j$, the $k$ loop then looks for a value which minimises the cost. For example, if $i = 2$ and $j = 5$:

\[
\begin{align*}
m[2, 2] + m[3, 5] + (p_1 \cdot p_2 \cdot p_5) &= 0 + 2500 + (35 \cdot 15 \cdot 20) = 13000 \\
m[2, 3] + m[4, 5] + (p_1 \cdot p_3 \cdot p_5) &= 2625 + 1000 + (35 \cdot 5 \cdot 20) = 7125 \\
m[2, 4] + m[5, 5] + (p_1 \cdot p_4 \cdot p_5) &= 4375 + 0 + (35 \cdot 10 \cdot 20) = 11375
\end{align*}
\]

From the $m$ table, we see that the lowest cost solution for the whole chain, i.e. where $i = 1, j = 6$, is $15125$. 
Matrix-chain Multiplication (13)

- From the $s$ table we can then reconstruct the optimal solution using a simple recursive algorithm since we have already computed all the overlapping sub-problems:

  \[
  \text{OUTPUT-ORDER}(s, i, j) \begin{align*}
  &\text{begin} \\
  &\text{if } i = j \text{ then} \\
  &\quad \text{OUTPUT("A"}_i) \\
  &\text{else} \\
  &\quad \text{OUTPUT("")} \\
  &\quad \text{OUTPUT-ORDER}(s, i, s[i, j]) \\
  &\quad \text{OUTPUT-ORDER}(s, s[i, j] + 1, j) \\
  &\quad \text{OUTPUT("")} \\
  &\text{end}
  \end{align*}
  \]

- For our example, OUTPUT-ORDER$(s, 1, 6)$ prints $(A_1(A_2A_3))(A_4A_5)A_6))$
Conclusions

The two examples represent two extremes of the dynamic programming spectrum:

- The Fibonacci sequence example makes the whole thing look like a trivial idea.
- The matrix chains example is quite involved but allows us to compute something that would normally be computationally hard.

The general properties that are needed are for dynamic programming are:

- Optimal substructure: an optimal solution to a problem contains optimal solutions to subproblems
- Overlapping subproblems: the total number of distinct subproblems which need solving is reasonably small
- We can implement dynamic programming recursively using memoisation or iteratively from the bottom up
- Other classic examples are computing the edit distance between two strings, the longest common subsequence and Floyd-Warshall’s all pairs shortest path algorithm
Further Reading

- **Introduction to Algorithms**
  - Chapter 15 – Dynamic Programming
  - Chapter 28 – Matrix Operations

- **Algorithm Design**
  J. Kleinberg and É.Tardos.
  - Chapter 6 – Dynamic Programming