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

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

A Simple Example (1)

- 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.

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:

\[
\text{FIBONACCI}(n) \begin{align*}
\text{begin} \\
\text{if } n = 0 \text{ then return } 0 \\
\text{if } n = 1 \text{ then return } 1 \\
\text{return } \text{FIBONACCI}(n - 1) + \text{FIBONACCI}(n - 2)
\end{align*}
\]

Note how the computation involved in the algorithm relates to our definition of dynamic programming:

- There are overlapping sub-problems, for example computing \( \text{FIBONACCI}(n - 1) \) overlaps \( \text{FIBONACCI}(n - 2) \).
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 (+) (\*\* tail) fibs
fastfib 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:

**Pre-computation**

```plaintext
f[0] ← 0
f[1] ← 1
```

**Main Algorithm**

```plaintext
FIBONACCI(n)
begin
for i = 2 upto n step 1 do
    f[i] ← f[i - 1] + f[i - 2]
return f[n]
end
```

- 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

Matrix-chain Multiplication (1)

- 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.

Matrix-chain Multiplication (2)

- 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 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 $n \geq 2$, $P(n) = P(1)P(n - 1) + P(2)P(n - 2) + \ldots + P(n - 1)P(1)$
  - 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 $P(n) \geq c \sum_{k=1}^{n-1} 2^n = c(n - 1)2^n \geq 2^n$
  - 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 (5)

- The first task is to characterise the structure of an optimal ordering.
- Say we have a chain of matrices $< A_i, A_{i+1}, \ldots, A_j >$. We denote the product of these matrices by $A_{i..j}$.
- 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 $A_{i..k}$ and $A_{k+1..j}$.
  - 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.

Matrix-chain Multiplication (6)

- 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 $A_{i..j}$:
  
  - 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.

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 \( p = (30, 35, 15, 5, 10, 20, 25) \).

Note that from this input \( p = (30, 35, 15, 5, 10, 20, 25) \).

The final algorithm reads as follows:

```java
public static void main(String[] args) {
    String[] p = new String[]{"1", "2", "3", "4", "5", "6", "7"};
    System.out.println(calculateOrder(p));
}

public static String calculateOrder(String[] p) {
    int n = p.length;
    int[][] m = new int[n][n];
    int[][] s = new int[n][n];
    for (int i = 0; i < n; i++) {
        m[i][i] = 0;
        s[i][i] = i;
    }
    for (int l = 1; l < n; l++) {
        for (int i = 0; i < n - l; i++) {
            int j = i + l;
            m[i][j] = Integer.MAX_VALUE;
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j];
                if (q < m[i][j]) {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    return Arrays.toString(s);
}
```

It doesn't take too much effort to see that it is \( O(n^3) \).
Matrix-chain Multiplication (11)

- From this list of matrices, we produce \( m \) and \( s \) tables:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 \\
15750 & 2625 & 750 & 1000 & 5000 & \\
11875 & 10500 & 3500 & \\
5375 & 4375 & 2500 & 3500 & \\
3750 & 2500 & 5000 & \\
3500 & 3500 & \\
15750 & 2625 & 750 & 1000 & 5000 & \\
11875 & 10500 & \\
5375 & \\
3750 & 2500 & 5000 & \\
3500 & \end{array}
\]

- 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.

Matrix-chain Multiplication (12)

- 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 \( j = 5 \):
    \[
    m[2, 5] = \min \left\{ m[2, 2] + m[2, 5] + (p_1 \cdot p_2 \cdot p_3), \quad m[2, 4] + m[4, 5] + (p_2 \cdot p_3 \cdot p_4), \quad m[2, 5] + m[5, 5] + (p_3 \cdot p_4 \cdot p_5) \right\}
    \]
    \[
    = 0 + 2500 + (35 \cdot 15 \cdot 20) = 13000
    \]
  - 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:

```plaintext
OUTPUT-ORDER(s, i, j)
begin
if i = j then
OUTPUT("A")
else
OUTPUT(" (")
OUTPUT-ORDER(s, i, s[i, j])
OUTPUT-ORDER(s, s[i, j] + 1, j)
OUTPUT(")")
end
```

- For our example, OUTPUT-ORDER(s, 1, 6) prints ((A1(A2A3))(A4A5)A6))

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