Dynamic Programming

– Formalized by Richard Bellman

– "Programming" relates to use of tables, rather than computer programming.

Fibonacci Numbers

$F_1 = F_2 = 1$.
$F_n = F_{n-1} + F_{n-2}$, for $n \geq 3$.

Fib(n)

If $n = 1$ or $n = 2$, then return 1

Else return $Fib(n-1) + Fib(n-2)$

End

Problem:
Too time consuming.

Time complexity: Exponential

$$\begin{array}{rcl} T(n) & = & T(n-1)+T(n-2)+constant \\ & > & 2T(n-2), \\ \Omega(2^{n/2}) & & \end{array}$$

Calculates $Fib(n-2)$ twice, $Fib(n-3)$ thrice, $Fib(n-4)$ five times, and so on.
We next consider a method which memorizes the values already computed.

Fib(n)

For $i = 3$ to n, { set $memF(i) = -1$ }

$memF(1) = memF(2) = 1$.

FibMod(n)

End

FibMod(n)

If $memF(n) \neq -1$, then return $memF(n)$.

Else

{ $memF(n) = FibMod(n-1) + FibMod(n-2)$.

Return $memF(n)$. }

End

$T(n) = T(n-1) + O(1)$
(note that, for computing $T(n-2)$, which was memorized, it takes time $O(1)$).

Thus,
$T(n) = O(n)$

Further simplification:

Fib(n):

$F(1) = 1$

$F(2) = 1$

For $i = 3$ to n {

$F(n) = F(n-1) + F(n-2)$

}

Return $F(n)$

End

Fib(n):

If $n = 1$ or $n = 2$, then return 1

Else

    prevprev=1

    prev=1

    For $i = 3$ to n {

        $f = prev + prevprev$

        $prevprev = prev$

        $prev = f$

    }

    Return f

End

Coin changing using dynamic programming

Given some denominations $d[1] > d[2] > \ldots > d[n] = 1$.
To find the minimal number of coins needed to make change for certain amount S.
Greedy algorithm was not optimal for some denominations.
We will give a dynamic programming algorithm which is optimal for all denominations.

Intuition:
Let $C[i, j]$ denote the number of coins needed to obtain value j, when one is only allowed to use coins $d[i], d[i+1], \ldots, d[n]$.
Then, $C[n, j] = j$, for $0 \leq j \leq S$.

Computing: $C[i-1, j]$:

If we use at least one coin of denomination $d[i-1]$:
$1 + C[i-1, j - d[i-1]]$

If we do not use any coins of denomination $d[i-1]$:
$C[i, j]$

Taking minimum of above, we get:
$C[i-1, j] = min(1 + C[i-1, j - d[i-1]], C[i, j])$

For $j = 0$ to S { $C[n, j] = j$ }

For $i = n$ down to 2 {

    For $j = 0$ to S {

        If $j \geq d[i-1]$, then

        $C[i-1, j] = min(1 + C[i-1, j - d[i-1]], C[i, j])$

        Else $C[i-1, j] = C[i, j]$

    }

}

Complexity: $O(S * n)$

To give coins used:

For $j = 0$ to S { $C[n, j] = j$; $used[n, j]$=true }

For $i = n$ down to 2 {

    For $j = 0$ to S {

        If $j \geq d[i-1]$ and $1 + C[i-1, j-d[i-1]] < C[i, j]$, then

            $\{C[i-1, j] = 1 + C[i-1, j-d[i-1]];$

              $used[i-1, j] = true\}$

        Else { $C[i-1, j] = C[i, j]$;

            $used[i-1, j] = false$ }

    }

}

For $i = 1$ to n { $U[i] = 0$ }

$i = 1$

$val = S$

While $val \geq 0$ {

    If $used(i, val) = true$, then $U[i] = U[i] + 1$, $val = val - d[i]$

    Else $i = i + 1$

}

Now $U[i]$ gives the number of coins with denomination $d[i]$ used.

Matrix Multiplication

Suppose we have matrices M_j with r_j rows and c_j columns, for $1 \leq j \leq k$, where $c_j = r_{j+1}$, for $1 \leq j < k$

We want to compute $M_1 \times M_2 \times \ldots \times M_k$.
Note that matrix multiplication is associative (though not commutative). So whether we do the multplications (for $k = 3$) as
$(M_1 \times M_2) \times M_3$
$M_1 \times (M_2 \times M_3)$
does not matter for final answer.

Note that multiplying matrix of size $r \times c$ with matrix of size $c \times c'$ takes time $r \times c \times c'$ (for standard method).

Different orders of multiplication can give different time complexity for matrix chain multiplication!

If $r_1 = 1$, $c_1 = r_2 = n$, $c_2 = r_3 = 1$ and $c_3 = n$, then

Doing it first way $((M_1 \times M_2) \times M_3)$ will give complexity $r_1c_1c_2 + r_1c_2c_3 = 2n$

Doing it second way $(M_1 \times (M_2 \times M_3))$ will give complexity $r_2c_2c_3 + r_1c_1c_3 = 2n^2$

Given n martices $M_1, M_2, \ldots, M_n$,
we want to find $M_1 \times M_2 \times \ldots \times M_n$
but minimize the number of multiplications (using standard method of multipliation).

Let $F(i, j)$ denote the minimum number of operations needed to compute $M_i \times M_{i+1} \times \ldots \times M_j$.
Then, $F(i, j)$, for $i < j$, is minimal over k (for $i \leq k < j$) of
$F(i, k) + F(k + 1, j) + cost(M_i \times \ldots \times M_k, M_{k+1} \times \ldots \times M_j)$
Here $cost(M_i \times \ldots \times M_k, M_{k+1} \times \ldots \times M_j) = r_i c_k c_j$

To compute $F(i, j)$ for $1 \leq i \leq j \leq n$,

$F(i, i) = 0$, for $1 \leq i \leq n$.

$F(i, j)$, for $1 \leq i < j \leq n$, by using formula given earlier.

Need appropriate order of computing $F(i, j)$ so that we do not duplicate work: what is needed is available at the time of use.

Do it in increasing order of $j - i$.

1. For $i = 1$ to n { $F(i, i) = 0$ } (* Initialization *)
2. For $r = 1$ to $n - 1$ {
3. For $i = 1$ to $n - r$ {
4. $j = i + r$.
   5. $F(i, j) = min_{k=i}^{j-1}[F(i, k) + F(k + 1, j) + r_i c_k c_j]$.
   }
}

Complexity of finding the optimal order: $O(n^3)$
($O(n^2)$ values to be computed, and computation at step 5 takes time $O(n)$).

Note: This is complexity of finding the optimal solution, not the complexity of matrix multiplication!

To determine the exact order of multiplication needed, one can do it by keeping track of the k which was used for $F(i, j)$. That will give us the order of multiplication.

Dynamic Programming Algorithms in general

– Define sub problems
– use optimal solutions for "sub problems" to give optimal solutions for the larger problem.
– using optimal solutions to smaller problems, one should be able to determine an optimal solution for a larger problem.

Note: We do not just combine solutions for arbitrary subproblems. For example, if we use coin denominations 1, 5 and 50, and find optimal solution for $S = 6$ and $S = 4$, then respectively, we will get $U_6[1] = 1$, $U_6[5] = 1$ and $U_4[1] = 4$ respectively. However, just combining them for U_{10} will give $U_{10}[1] = 5$ and $U_{10}[5] = 1$, which is not the optimal solution.

So, we "use" optimal solutions for some specific subprobelms to obtain an optimal solution for the larger problem.
– The subproblems are often generated by "reducing" some parts of the original problem
– ordering among subproblems, so that result of smaller subproblems are available when solving larger subproblem.

Optimal substructure Property:
– We always need that for the optimal solution, the "reduced part" is optimal for the smaller problem.

Example: If S is optimal solution to coin changing problem for value S, then if remove one coin, with denomination d, from S, then it is optimal solution for $S - d$.

Example: Given a weighted graph consider the problem of finding the longest simple path. Then, this does not satisfy the optimal substructure property.

If path $v_0, v_1, v_2, \ldots, v_k$ is longest simple path from v_0 to v_k, then it does not mean that $v_0, v_1, \ldots, v_{k-1}$ is the longest simple path from v_0 to v_{k-1}!

When do we use dynamic programming?
When, the sub-problems are overlapping, dynamic programming allows one to avoid duplication of work as compared to recursion.