Dynamic Programming

- Formalized by Richard Bellman
- "Programming" relates to planning/use of tables, rather than computer programming.
- Solve smaller problems first, record solutions in a table; use solutions of smaller problems to get solutions for bigger problems.
- Differs from Divide and Conquer in that it stores the solutions, and subproblems are "overlapping"
- Fibonacci Numbers, Binomial Coefficients, Warshall's algorithm for transitive closure, Floyd's all pairs shortest path, Knapsack, optimal binary search tree

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.
  The sub problems $Fib(n-1)$ and $Fin(n-2)$ are 'overlapping'
  Time complexity: Exponential

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

- 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.

Dynamic Programming Algorithm

Fib(n):

$F(1) \leftarrow 1$

$F(2) \leftarrow 1$

For $i = 3$ to n

$F(n) \leftarrow F(n-1) + F(n-2)$

EndFor

Return $F(n)$

End

- $T(n) \in O(n)$.
- Uses Extra Memory.

Using constant memory

Fib(n):
If $n = 1$ or $n = 2$, then return 1
Else

$prevprev \leftarrow 1$

$prev \leftarrow 1$

For $i = 3$ to n $\{$

$f \leftarrow prev + prevprev$

$prevprev \leftarrow prev$

$prev \leftarrow f$

$\}$

Return f

End

Binomial Coefficients

- $C(n,k) = {}^nC_k = \binom{n}{k}$: number of ways to choose k out of n objects
- $C(n,k) = C(n-1,k-1) + C(n-1,k)$, for $n > k > 0$.
- $C(n,0) = 1 = C(n,n)$.
- So $C(n,k)$ can be computed using smaller problems.

Binomial Coefficients

	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Complexity for calculating $C(n, k)$ (besides the initial $C(i, 0), C(i, i)$).

$\Sigma_{i=1}^{k} \Sigma_{j=1}^{i-1} 1 + \Sigma_{i=k+1}^{n} \Sigma_{j=1}^{k} 1$

$= (k-1)(k)/2 + k(n-k) \in \Theta(nk)$.

Transitive Closure

- Intuitively, a relation is transtive if aRb and bRc implies aRc.
- Think of a directed graph. Then, transitive closure can be used to determine if there is a non-trivial directed path from node i to node j.

Warshall's Algorithm

- Given adjacency matrix A
- $R^k(i,j)$ denotes if there is a path from i to j which has intermediate vertices only among $\{1,2,\ldots,k\}$.
- Thus, $R^0(i,j) = A(i,j)$.
- $R^{k+1}(i,j)$ from $R^k(i,j)$?
- If $R^k(i,j) = 1$, then $R^{k+1}(i,j) = 1$
- If $R^k(i,k+1) = 1$ and $R^k(k+1,j) = 1$, then $R^{k+1}(i,j) = 1$
- Thus, we compute $R^0, R^1, \ldots$ one by one.
- Can consider this as 3D matrix (involving the parameters i,j,k)
- Takes time $\Theta(n^3)$.

Warshall's Algorithm

Input $A[1..n, 1..n]$.

$R[0, i, j] = A[i, j]$.

For $k = 1$ to n do

For $i = 1$ to n do

For $j = 1$ to n do

If $R[k-1, i, j] = 1$ or $(R[k-1, i, k] = 1$ and $R[k-1, k, j] = 1)$,

then $R[k, i, j] \leftarrow 1$ Else $R[k, i, j] \leftarrow 0$ Endif

Endfor

Endfor

Endfor

Complexity: $\Theta(n^3)$.

All pairs shortest path algorithm

- Suppose A is the adjacency matrix of a weighted graph, that is, $A[i, j]$ gives the weight of the edge (i, j).
- Here the vertices are numbered 1 to n.
- Here we assume that $A[i, j]$ is non-negative. One can handle negative weights also as long as there is no negative circuits (because going through negative circuits repeatedly can reduce the weight of the path by arbitrary amount).
- If edge does not exist then the weight is taken to be ∞.
- We want to find shortest path between all pairs of vertices.

Floyd's algorithm

- Idea similar to Warshall's algorithm.
- $D_k[i, j]$ denote the length of the shortest path from i to j where the intermediate vertices (except for the end vertices i and j) are all $\leq k$.
- $D_0[i, j] = A[i, j]$.
- Here we assume $A[i, i] = 0$ for all i.
  If initially not so, then update $A[i, i]$ to be so.
- To find, $D_k[i, j]$, for $1 \leq k \leq n$:
- $D_k[i, j] = min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$.

- In the algorithm, $next[i,j]$ denotes the vertex which appears just after i in the shortest (known) path from i to j.
- Note: $D_k[i,k] = D_{k-1}[i,k]$ and $D_k[k,j] = D_{k-1}[k,j]$. So, we can do the computation in place! (using D itself to compute $D_0, D_1, D_2, \ldots$).

Floyds algorithm

Input $A[1..n, 1..n]$

For $i = 1$ to n do, For $j = 1$ to n do

$next[i, j] = j$; $D[i, j] = A[i, j]$

EndFor EndFor

For $i = 1$ to n do $D[i, i] = 0$ EndFor

For $k = 1$ to n do

For $i = 1$ to n

For $j = 1$ to n

If $D[i, j] > D[i, k] + D[k, j]$, then

$D[i, j] = D[i, k] + D[k, j]$

$next[i, j] = next[i, k]$

Endif

EndFor

EndFor

EndFor

$Path(i, j)$

$current = i$;

Print(i);

While $current \neq j$ do

  $current = next(current, j)$;

  Print($current$)

EndWhile

$0/1$ **Knapsack problem**

- Some objects $O_1, O_2, \ldots, O_n$
- Their weights $W_1, W_2, \ldots, W_n$ (assumed to be integral values)
- Their values $V_1, V_2, \ldots, V_n$
- Capacity C
- To find a set $S \subseteq \{1, 2, \ldots, n\}$ such that
  $\Sigma_{i \in S} W_i \leq C$ (capacity constraint)
  and
  $\Sigma_{i \in S} V_i$ is maximsed
- Note that here we cannot use fractional items! We either take the whole or nothing of each item.

- Let $F(C, j)$ denote the maximum value one can obtain using capacity C such that objects chosen are only from $O_1, \ldots, O_j$.
- Then, $F(C, 0) = 0$.
- If $W_j \leq C$, then $F(C, j) = max(F(C, j-1), F(C - W_j, j-1) + V_j)$.
- If $W_j > C$, then $F(C, j) = F(C, j-1)$.

For $s = 0$ to C do

$F(s, 0) = 0$

EndFor.

For $s = 1$ to C do

For $j = 1$ to n do

If $W_j \leq s$, then

$F(s, j) = max(F(s, j-1), F(s-W_j, j-1)+V_j)$;

Else $F(s, j) = F(s, j-1)$ Endif

EndFor

EndFor

Complexity: $O(C * n)$

Want to see which objects are chosen

For $s = 0$ to C do $F(s,0) = 0; Used(s,0) = false$ EndFor
    For $s = 1$ to C do
        For $j = 1$ to n do
            If $W_j \leq s$, then
                $F(s,j) =$
                    $max(F(s,j-1), F(s-W_j, j-1) + V_j)$;
                If $F(s,j) = F(s,j-1)$, then
                    $Used(s,j) = false$
                Else $Used(s,j) = true$ Endif
            Else $F(s,j) = F(s,j-1)$; $Used(s,j) = false$
            Endif
        EndFor
    EndFor

$Left = C$

For $j = n$ down to 1 do $\{$

    If $Used(Left, j) = true$, then

        Print("Pick item" j); $Left = Left - W_j$;

EndIf

EndFor

Coin changing using DP

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

CoinChange
For $j = 0$ to S do
    $C[n, j] = j$
Endfor
For $i = n$ down to 2 do
    For $j = 0$ to S do
        If $j \geq d[i-1]$, then
        $C[i-1, j] = min(1 + C[i-1, j - d[i-1]], C[i, j])$
        Else $C[i-1, j] = C[i, j]$
    EndFor
EndFor

Complexity: $\Theta(S * n)$

To give coins used:

For $j = 0$ to S do $C[n, j] \leftarrow j$; $used[n, j] \leftarrow true$ EndFor
For $i = n$ down to 2 do
    For $j = 0$ to S do
        If $j \geq d[i-1]$ and $1 + C[i-1, j-d[i-1]] < C[i, j]$,
          then
            $C[i-1, j] \leftarrow 1 + C[i-1, j-d[i-1]]$;
            $used[i-1, j] \leftarrow true$
        Else $C[i-1, j] \leftarrow C[i, j]$;
            $used[i-1, j] \leftarrow false$
    EndFor
EndFor

For $i = 1$ to n do

$\quad U[i] \leftarrow 0$

EndFor

$i \leftarrow 1$

$val \leftarrow S$

While $val \geq 0$ do

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

$\quad$ Else $i \leftarrow i + 1$

EndWhile

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 $r_1 c_1 c_2 + r_1 c_2 c_3 = 2n$
- Doing it second way ($M_1 \times (M_2 \times M_3)$) will give complexity $r_2 c_2 c_3 + r_1 c_1 c_3 = 2n^2$

- 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 $1 \le i \le j \le n$,
- $F(i,i) = 0$, for $1 \le i \le n$.
- $F(i,j)$, for $1 \le i < j \le n$, 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 $j - i$.

For $i = 1$ to n do

$F(i, i) = 0$

EndFor

For $r = 1$ to $n - 1$ do

3. For $i = 1$ to $n - r$ do

4. $j = i + r$.

5. $F(i, j) = min_{k=i}^{j-1}[F(i, k) + F(k + 1, j) + r_i c_k c_j]$.

EndFor

EndFor

Complexity

- 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.

Dyn Prog 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.

Longest Common Subsequence

- Motivated by problem of whether two proteins are similar.
- A subsequence of a sequence $a[1]a[2]\ldots a[n]$ is a sequence of the form $a[i_1]a[i_2]\ldots a[i_k]$ such that $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.
- Example: $ACAE$, AB, BA, AE, E are subsequences of $ABCADE$.
- DA is not a subsequence of $ABCADE$.
- Longest common subsequence of two sequences ABC and BDC is BC.

- Input: $x[1]x[2]\ldots x[n]$ and $y[1]y[2]\ldots y[m]$
- Find the longest common subsequence of the above two sequences (strings).
- If $x[n] = y[m]$, then the longest common subsequence is:
  (the longest common subsequence of
  $x[1]\ldots x[n-1]$ and $y[1]\ldots y[m-1]$)
  followed by $x[n]$
- If $x[n] \neq y[m]$, then the longest common subsequence is:
  (the longest common subsequence of $x[1]\ldots x[n-1]$ and $y[1]\ldots y[m]$) or
  (the longest common subsequence of $x[1]\ldots x[n]$ and $y[1]\ldots y[m-1]$)

- $F(i, j)$ length of the longest common subsequence of $x[1]x[2] \dots x[i]$ and $y[1]y[2] \dots y[j]$, where $0 \leq i \leq n$ and $0 \leq j \leq m$
- Base Case: $F[i, j] = 0$, for $i = 0$ or $j = 0$
- Solve the recurrence equation given earlier to obtain other $F[i, j]$.

$LCS(x[1]\ldots x[n], y[1]\ldots y[m])$

For $i = 0$ to n do $F(i, 0) = 0$ EndFor

For $j = 0$ to m do $F(0, j) = 0$ EndFor

For $i = 1$ to n

For $j = 1$ to m

If $x[i] = y[j]$, then $F(i, j) = 1 + F(i - 1, j - 1)$

Endif

If $x[i] \neq y[j]$, then

$F(i, j) = max(F(i - 1, j), F(i, j - 1))$ Endif

EndFor

EndFor

End

Complexity: $\Theta(mn)$

LCSP$(x[1] \dots x[n], n, m, F)$

If $F(n,m) = 0$, then return.

ElseIf $F(n,m) = F(n,m-1)$, then return $LCSP(x,n,m-1,F)$

ElseIf $F(n,m) = F(n-1,m)$, then return $LCSP(x,n-1,m,F)$

Else return $LCSP(x,n-1,m-1,F) \cdot x(n)$

End

The main idea of dynamic programming is:

- Solve several smaller problems
- Store these solutions (in an array) and use them to solve large problems
- Have a recurrence; solve in particular order
- Subproblems have overlaps (i.e., some of the subsubproblems they solve are same)
- In Divide an conquer usually do not have overlaps, and do not store the solutions.