Algorithm Design Strategies:

- Design by Induction
- Divide and Conquer
- Dynamic Programming

Design by Induction:

The principle of design by induction is as follows:

Assume that I know how to solve a problem of a given size. How do I express the solution to problem of a larger size in terms of the solution to problem of known size.

Design by Induction: Evaluating a polynomial.

Consider the polynomial:

$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

Assume that you know how to compute $P_{n-1}(x)$. We know that $P_0(x) = a_0$.

Given $P_{n-1}(x)$, we can easily compute $P_n(x)$ as follows:

$P_n(x) = P_{n-1}(x) + a_n x^n$

While we can write a program in this manner, computing $P_n(x)$ from $P_{n-1}(x)$ still requires computation of $a_n x^n$ followed by an addition. This takes n multiplications for computing $a_n x^n$ followed by 1 addition.

It is easy to verify that computing $P_0, P_1, P_2, ..., P_n$, in series therefore requires $O(n^2)$ computations.

Can we do better than this?

Design by Induction: Evaluating a polynomial.

Consider, again, the polynomial:

$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

Assume that you know how to compute $P_{n-1}(x)$ and $S_{n-1}(x)$. Here, $S_{n-1}(x) = x^{n-1}$.

We know that $P_0(x) = a_0$ and $S_0(x) = 1$.

Now we can write:

$S_n(x) = S_{n-1}(x) * x$
$P_n(x) = P_{n-1}(x) + a_n S_n(x)$

It is easy to see that computing $(P_n(x), S_n(x))$ from $(P_{n-1}(x), S_{n-1}(x))$ requires 2 multiplications and 1 addition.

It is easy to verify that computing $P_0, P_1, P_2, ..., P_n$, in series therefore requires $2n$ multiplications and n additions.

Can we do yet better?

Design by Induction: Evaluating a polynomial.

Consider, again, the polynomial:

$p_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

This time, we define

$P'_1(x) = a_n x + a_{n-1}$
$P'_2(x) = (a_n x + a_{n-1}) * x + a_{n-2}$ *or* $P'_1(x)*x + a_{n-1}$

$P'_n(x) = P'_{n-1}(x)*x + a_0$

It is easy to see that $P_n(x)$ is the same as $P'_n(x)$.

Also, computing $P'_i(x)$ from $P'_{i-1}(x)$ takes one addition and one multiplication.

However, in this case, computing in sequence $P'_1(x), P'_2(x), ..., P'_n(x)$ only takes n additions and n multiplications. While this is still $O(n)$, this algorithm is twice as fast as the preceding one.

Dynamic Programming (DP):

- Similar to Divide and Conquer (DC).
- Express solution of a problem in terms of solutions to subproblems.
- Key difference between DP and DC is that while subproblems in DC are independent, subproblems in DP may themselves share subsubproblems. This means that if these were treated as independent subproblems, the complexity would be higher.
- DP is typically used to solve optimization problems.
- In bioinformatics, the most common use of DP is in sequence matching and alignment.

DP works on the Bellman principle of optimality, namely, DP will return a desired solution for problems where the desired (optimal) solution to a problem can be composed from desired (optimal) solution to subproblems.

Dynamic Programming: Optimal matrix multiplication parenthesization.

Consider the problem of computing the product of the matrices $A_1.A_2.A_3... . A_n$.

Here, each A_i is of dimension $r_i \times r_{i+1}$.

Clearly, the product is a matrix of dimension $r_1 \times r_{n+1}$.

The sequence in which the product is computed critically impacts the number of operations. Recall that multiplying two matrices of dimensions $i \times j$ and $j \times k$ takes $i \times j \times k$ operations (each operation is a single add and multiply).

For example, if A_1 is of dimension 3 x 3, A_2 of dimension 3 x 3 and A_3 of dimension 3 x 1,

$(A_1.A_2).A_3$ takes 3 x 3 x 3 + 3 x 3 x 1 = 36 operations.

on the other hand,

$A_1.(A_2.A_3)$ takes 3 x 3 x 1 + 3 x 3 x 1 = 18 operations.

The objective of the optimal matrix parenthesization problem is to derive the parenthesization that yields the lowest operation count.

Dynamic Programming: Optimal matrix multiplication parenthesization

Let $C(i, j)$ denote the number of operations in an optimal parenthesization (best multiplication order) of matrices $A_i.A_{i+1}.A_{i+2}... . A_j$.

Now, if $j > i$, there must exist a k such that

$A_i.A_{i+1}.A_{i+2}... . A_j = (A_i.A_{i+1}... . A_k).(A_k.A_{k+1}... . A_j)$

We know, by definition that the optimal parenthesization of $(A_i.A_{i+1}... . A_k)$ requires $C(i,k)$ operations and that of $(A_k.A_{k+1}... . A_j)$ requires $C(k,j)$ operations.

Therefore, the optimal number of operations to compute $A_i.A_{i+1}.A_{i+2}... . A_j$ must be defined as the minimum over all k of $C(i,k) + C(k,j) +$ (operations required to multiply $(A_i.A_{i+1}... . A_k)$ and $(A_k.A_{k+1}... . A_j)$).

Therefore the following recurrence relation follows:

$$C(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min (C(i,k) + C(k,j) + (r_i . r_k . r_{j+1}))_{k=i, i+1, .. j} \end{cases}$$

This recurrence relation can be used to compute the optimal parenthesization.

Dynamic Programming: Optimal matrix multiplication parenthesization

We illustrate the algorithm with the following example:

			15125		
		11875	10500		
	9375	7125	5375		
7875	4375	2500	3500		
15750	2625	750	1000	5000	
0	0	0	0	0	0
A_0	A_1	A_2	A_3	A_4	A_5
30x35	35x15	15x5	5x10	10x20	20x25

The optimal parenthesization of the six matrices can be done in 15125 operations. This is less than just multiplying A_0 with A_1!

Dynamic Programming: Longest Common Subsequence.

A sequence Y is a subsequence of X if Y can be derived by deleting 0 or more entries in X.

For example, <A, G, D> is a subsequence of
<B, A, C, D, G, A, D>

On the other hand, <A, G, D> is not a subsequence of
<B, A, C, D, G, A>

The objective of the longest common subsequence problem is, given two sequences X and Y, to find a sequence Z which is the longest subsequence of X and Y.

For example,

X : < A, B, C, B, D, A, D>
Y: < B, D, C, A, B, A>

Here, < B, C>, < B, A>, and < B, C, B, A> are all subsequences (among others), however, as we shall show soon, <B, C, B, A> is a longest common subsequence of X and Y.

Dynamic Programming: Longest Common Subsequence.

The key to the DP algorithm for longest common subsequence is as follows:

Say *C(i, j)* is the length of the longest common subsequence of first *i* characters of X and first *j* characters of Y. Now, say, we have somehow computed the longest common subsequence of first *i-1* characters of X and *j-1* characters of Y, i.e., we know *C(i-1, j-1)*. We need to use this information to compute *C(i, j)*.

In this case, if the *i*th character of X matches the *j*th character of Y, then *C(i, j)* must be *C(i-1, j-1) + 1* (since the length can be increased because of the matching character).

The interesting case happens when the *i*th character of X does not match *j*th character of Y. In this case, we have one of two alternatives -- we discard the *i*th character of X and carry on, or we discard the *j*th character of Y and carry on. In the former case, *C(i, j) = C(i-1, j)* and in the latter case, *C(i, j) = C(i, j-1)*. Since we are looking for the maximum length sequence, this leads to the simple recurrence relation:

$$C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C(i-i, j-1) + 1 & \text{if } X_i = Y_i \\ \max(C(i, j-1), C(i-1, j)) & \text{if } X_i \mathrel{!}= Y_i \end{cases}$$

This recurrence relation can be used to solve the longest common subsequence problem.

Dynamic Programming: Longest Common Subsequence.

Example:

Consider two sequences:
X : < A, B, C, B, D, A, D>
Y: < B, D, C, A, B, A>
The DP algorithm constructs the *C* table as follows:

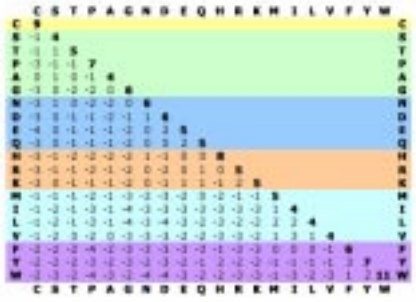
Table 1: C table for matching X and Y.

	Y	B	D	C	A	B	A
X	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	1	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
D	0	1	2	2	3	4	4

We can identify longest common subsequence by locating all diagonal transitions in the C table. In this case, there are multiple longest common subsequences. One is <B, C, B, A>. What are the others?

The Smith-Waterman Algorithm for Matching.

The Smith Waterman algorithm augments the matching process with similarity tables and gap penalties. The idea is that not all amino-acids (characters) are equally similar or dissimilar. An example of a similarity table is Blosum62 shown below:



This table indicates that if you match a C in X with a C in Y, you can add 9 to the score. However, if you match an S in X with an S in Y, you only add 4 to your score. Similarly, if you match a P in X with a C in Y, you must deduct 3 from your score. This is in contrast to the earlier scheme in which you added 1 if there is a match and did not do anything if there is a mismatch.

Smith-Waterman Matching: Example.

Table 1: Similarity Table

	H	E	A	G	A	W	G	H	E	E
P	-2	-1	-1	-2	-1	-4	-2	-2	-1	-1
A	-2	-1	4	0	4	-3	0	-2	-1	-1
W	-2	-3	-3	-2	-3	11	-2	-2	-3	-3
H	8	0	-2	-2	-2	-2	-2	8	0	0
E	0	5	-1	-2	-1	-3	-2	0	5	5
A	-2	-1	4	0	4	-3	0	-2	-1	-1
E	0	5	-1	-2	-1	-3	-2	0	5	5

Table 2: Smith-Waterman Match

		H	E	A	G	A	W	G	H	E	E
	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	4	2	0	0	0	0	0	0
W	0	0	0	2	2	0	11	9	7	5	3
H	0	8	6	4	2	0	9	9	17	15	13
E	0	6	13	11	9	7	7	7	15	22	20
A	0	4	11	17	15	13	11	9	13	20	21
E	0	2	9	15	15	14	12	10	11	18	19

Sequence 1: WGHE, Sequence 2: W_HE, Score: 22.