

Dynamic Programming

1 Assumptions

1. $O()$ notation etc.
2. DNA, RNA.

2 Dynamic Programming

Dynamic Programming has proven to be a very popular technique in Biological Sequence Analysis, exemplified by the Smith Waterman algorithms for sequence alignment. Problems that allow a Dynamic Programming solution have a few important characteristics: there is an objective that needs to be optimized, the problem can be decomposed recursively into sub-problems of a similar form, and the solution of each sub-problem is required many times in the course of obtaining the final solution. For such problems, Dynamic Programming can be used to solve each sub-problem once, and its result is then stored in a table for future use. We illustrate with an example:

Optimum Warehouse Expansion: *Tony is the logistics manager for a struggling internet retailer. Business is picking up again, finally, and Tony is charged with expanding warehouse storage to keep transactions smooth. Being cost-conscious in this new economy, he obtains a log of all recent warehouse shipments. The data is described schematically in Figure 1. Positive numbers refer to shipments that have arrived at the warehouse, and negative numbers refer to requests for outgoing shipments. If an item is requested when not available, the request is denied, and not applied to future incoming shipments. Those must be stored in the warehouse to wait for subsequent requests. Can Tony determine the minimum capacity needed to store all of the supplies before they are shipped.*

Let the array S be used to denote the shipment-log, with $S[i]$ representing the i 'th transaction. Also, let $C(i, j)$ denote the capacity needed to process transactions i through transactions j . As $C(i, j)$ cannot be negative, we have

$$C(i, j) = \max\{0, \sum_{k=i}^j S[k]\} \quad (1)$$

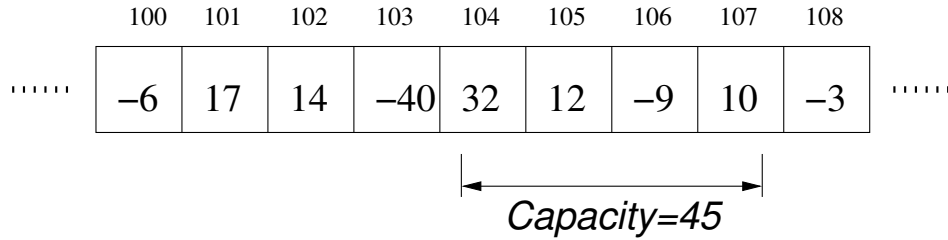


Figure 1: A log of Warehouse shipments. As an example, transaction 102 refers to an incoming shipment of 14 items, while transaction 103 is a request for 40 items to be shipped out. The maximum capacity needed is the maximum value of the net shipment over a range of transactions. Thus the capacity needed for transactions 104 through 108 is 45.

The shipment log therefore helps compute the required warehouse capacity C^W as

$$C^W = \max_{i,j} C(i, j) \tag{2}$$

A simple minded approach to computing C^W is described in Figure 2. Procedure *Warehouse-Capacity* is an $O(n^3)$ algorithm to compute C^W , where n is the size of the shipment-log. This could be prohibitive for large n . Certainly, we should do

```

Procedure Warehouse-Capacity()
   $C^W = 0$ 
  for  $i = 1$  to  $n$ 
    for  $j = i + 1$  to  $n$ 
      SUM =  $\max\{0, \sum_{k=i}^j S[k]\}$ 
      if (SUM >  $C^W$ )
         $C^W = \text{SUM}$ 
    end for
  end for

```

Figure 2: A simple $O(n^3)$ algorithm for computing the Warehouse Capacity

able to do better. The problem satisfies at least one of the requirements of dynamic programming. There is an objective (namely $C(i, j)$) that must be maximized over all i, j . Note also that there is a lot of redundant computation in the procedure. For all $i_1 \leq i$, and $j_1 \geq j$, the computation of $C(i_1, j_1)$ includes the recomputation of $C(i, j)$. To speed things up using dynamic programming, we need to check if there is a recursive structure, and if the corresponding sub-problems need to be solved repeatedly. In this case, we have already made the implicit observation that

$$C(i, j) = \max\{0, S[j] + C(i, j - 1)\} \tag{3}$$

Certainly, if we stored all values of the function $C(i, j)$ in an array (let us call it $C[i, j]$), we only need $O(1)$ time to compute $C[i, j]$ for all i, j . In procedure *Fast-Warehouse-Capacity*() (see Figure 3), we use this idea to describe an $O(n^2)$ algorithm

for our problem. *Fast-warehouse-Capacity()* is a dynamic programming procedure.

```

Procedure Fast-Warehouse-Capacity()
 $C^W = 0$ 
for  $i = 1$  to  $n$ 
     $C[i, i] = 0$ 
    for  $j = i + 1$  to  $n$ 
         $C[i, j] = \max\{0, S[j] + C[i, j - 1]\}$ 
        if ( $C[i, j] > C^W$ )
             $C^W = C[i, j]$ 
    end for
end for

```

Figure 3: An $O(n^2)$ dynamic programming algorithm for computing the Warehouse Capacity

It maximizes an objective function by breaking up the problem in sub-problems with a recursive structure. The sub-problems must be solved in a bottom-up fashion so that when the value of a sub-problem is needed again, it can simply be looked up. Finally, all dynamic programming algorithms, including Procedure *Fast-Warehouse-Capacity()*, must have an initialization step to solve the 'base-case' of the recursion.

While there are many steps in describing a Dynamic Programming solution to a problem, deciding if the problem has the appropriate recursive structure is critical, and is the most creative part of the enterprise. Often in this Chapter, and elsewhere in the book, we will 'solve' a problem simply by describing the recursion, leaving the details to the reader.

While the Warehouse problem and its solution are somewhat simplistic, they reveal most of the characteristics of the dynamic programming methodology. There are some other issues that we haven't discussed yet. *Choosing the right sub-problems* is one of them. As we will see, Procedure *Fast-Warehouse-Capacity()* is not so fast after all. We selected the wrong set of sub-problems: there were just too many. To get a hint of why this might be, look at the recursive step again

$$C[i, j] = \max\{0, S[j] + C[i, j - 1]\} \quad (4)$$

Observe that index i is 'static'. All sub-problems seem to arise by decreasing only the right index. Can we take advantage of this? Define function $R(j)$ as the maximum of all capacity values that end in index j . In other words,

$$R(j) = \max_{i' < j} C(i', j) \quad (5)$$

Clearly, it is sufficient to compute $R(j)$ for all j , as the range that gives the maximum capacity must end at some j . For any j , either $R(j) = 0$, there is an index $i \leq j$ such that $R(j) = C(i, j)$. If $i < j$, then $R(j - 1) = C(i, j - 1)$ (Why?). Also, if

```

Procedure Faster-Warehouse-Capacity()
 $C^W = 0$ 
 $R[0] = 0$ 
for  $i = 1$  to  $n$ 
     $R[j] = \max\{0, S[j] + R[j - 1]\}$ 
    if ( $R[j] > C^W$ )
         $C^W = R[j]$ 
end for

```

Figure 4: An $O(n)$ dynamic programming algorithm for computing Warehouse Capacity

$i == j$, then it must be because $R(j - 1) == 0$. In other words $R(j)$ has a recursive structure defined by

$$R(j) = \max\{0, S[j] + R(j - 1)\} \quad (6)$$

By now, the recursion should be sufficient to obtain the actual algorithm. For completeness, an $O(n)$ algorithm computing the Maximum capacity is shown in Figure 4. It is illustrative to see how fast this is. If the log had a million (10^6) transactions, and our computing capacity could deal with 10^9 instructions in a second, it would require only a small fraction (0.001) of a second to compute the optimum capacity. This contrasts with *Fast-Warehouse-Capacity()*, which would need about 17 mins., and with *Warehouse-Capacity()* which will take all of 31 years, and probably cost Tony his job. Let us consider a little twist to the problem. Not only do we need the maximum capacity, but also the series of transactions (described by indices i, j) which lead to the maximum value of $R[j]$. It is a common theme in dynamic programming problems that we are interested not only in the optimal value, but in the range that returns the optimal value. A little book-keeping helps us solve this problem with little additional effort. We introduce the term $L(j)$, defined as follows

$$L(j) = \begin{cases} j + 1 & \text{if } (R(j) == 0) \\ i & i \leq j \ \& \ C(i, j) = R(j) = \max_{i' \leq j} C(i', j) \end{cases} \quad (7)$$

It suffices to compute $L(j)$, as the desired series of transactions is simply $(L(j), j)$, where $j = \arg \max_{j'} R(j')$. It is left as an exercise for the reader to (Problem ??) show that the following recurrence is correct for all $0 \leq j \leq n$.

$$L(j) = \begin{cases} j + 1 & \text{if } (j == 0) \text{ or } (R(j) == 0) \\ L(j - 1) & \text{otherwise} \end{cases} \quad (8)$$

3 Dynamic Programming in Bioinformatics

Dynamic Programming has had a profound influence on Bioinformatics. This is typified (but hardly limited) by its use in sequence alignment algorithms. Without

further ado, we jump into this area **CHANGE THIS**. Figure 5 shows a comparison of two proteins using BLAST, a popular tool for sequence retrieval and alignment. A glance at the first picture (Figure ??a) shows extensive similarity between the two sequences, with over 63% of the residues being identical. It might therefore come as a surprise that the top sequence is a mouse protein while the bottom sequence is from the fruit fly (*Drosophila melanogaster*). This conservation of protein sequence through over a 100 million years of evolution is remarkable and indicates a functional role for these proteins. In mouse, this so called *tubby* gene is a receptor that is expressed in brain and mice with the gene knocked out show a phenotype consistent with obesity. On the other hand, the fruit fly gene was predicted computationally from the genomic sequence, and no function has been directly ascribed to it. Based on the conservation of the two sequences, it would be a reasonable guess to extrapolate a functional role that is similar to *tubby*. Now see Figure 5b. This time the aligned regions are quite distant with only about 33% identity in the aligned residues. Even so, there are enough similarities and regions of conservation to confirm an evolutionary relationship between the two sequences. The second sequence this time comes from a plant, *A. thaliana*. Identification of distant homologs is critical in assigning functional roles to novel proteins. Furthermore, a study of the conserved regions highlights those residues in the protein, which are important for maintaining structure and/or function of the protein. This is the new face of Biology. The initial biological hypothesis is usually the result of a *computational* experiment. In this case, the computation is to generate a pairwise alignment.

3.1 Sequence Alignment

The objective of pairwise-alignment is to line up the two sequences so that identical or 'physiochemically similar' residues are lined up against each other. This is not always easy. As seen in Figure 5b, the number of identical residues might be small, and *gaps* have to be opened in sequences to align residues properly. Formally, an alignment of two sequences can be thought of as a matrix with two rows. The cells of each row contain a sequence with gaps interspersed in between. Additionally, every column has at least one non-gap character. Sequences that are evolutionarily close should align with a majority of the columns lining up identical symbols (*matches*), and few columns with *mismatches* and gaps. As gaps refer to insertions and deletion of nucleotides, we will also refer to these as *indels*, and use the two terms interchangeably. To illustrate this with a simple example, see Figures 6, and 7. Two candidate alignments (denoted by A_1 , and A_2) are shown for the DNA strings AGATCCAG and AATTACCTG. Alignment A_1 has 6 columns with matches, and 4 with mismatches and indels. The other (A_2) has 5 matches and 4 with mismatches and indels. To choose between different candidate alignments, we define a numerical *score* for an alignment. Let Σ denote the alphabet from which the strings are drawn. Thus, for DNA, $\Sigma = \{A, C, G, T\}$. We augment Σ to include the gap symbol '-' by defining $\Sigma' = \Sigma \cup \{-\}$. Next, define a function $\mathcal{S} : \Sigma' \times \Sigma' \rightarrow \mathcal{R}$ that assigns a score to a pair of symbols. Finally, let A be a $2 \times m$ matrix defining an alignment.

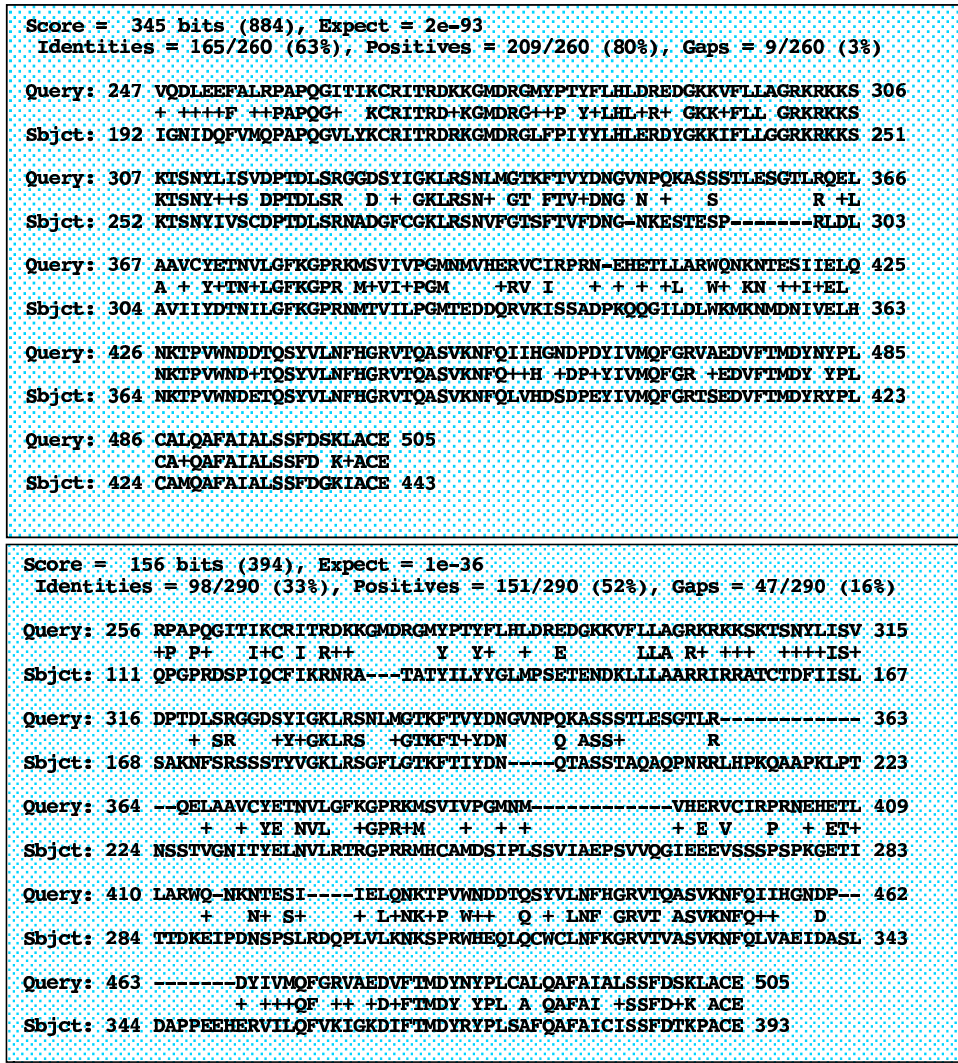


Figure 5: The tubby gene encodes a brain receptor and is involved in body weight regulation. (a) A pairwise sequence alignment of a portion of the Mouse *tubby* protein against an unknown protein in fruit-fly. (b) Another alignment of the tubby protein, this time with a plant, *Arabidopsis thaliana*. Each column of the alignment pairs up identical amino-acids, 'physiochemically similar' (denoted by a +), or dissimilar amino-acids, or aligns an amino-acid with a gap ('-').

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | G | A | T | - | - | C | C | A | G |
| A | - | A | T | T | A | C | C | T | G |

Figure 6: An alignment (Alignment $A1$) of the string AGATCCAG with AATTACCTG. The alignment has 10 columns, of which 6 are identical *matches*, 3 correspond to *indels*, i.e. insertion or deletion of characters, and 1 *mismatch*.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | G | A | T | - | C | C | A | G |
| A | A | T | T | A | C | C | T | G |

Figure 7: An alternative alignment ($A2$) of strings AGATCCAG and AATTACCTG. This alignment has 5 matches, 3 mismatches, and one indel

Alignment A is scored by score $\mathcal{S}(A[1, j], A[2, j])$ to each column j , and summing up scores for all columns. Many score functions are possible, and different functions may be chosen to model different biological problems. Consider for example,

$$\mathcal{S}_1(u, v) = \begin{cases} 2 & u = v \\ \infty & u = '-' \text{ \& } v = '-' \\ -1 & \text{otherwise} \end{cases} \quad (9)$$

In this case, alignment $A1$ in Figure 6 has a score of 8. This is superior to $A2$ with a score of 6. The situation changes if we change the score function to have a higher penalty for gapped columns. Consider

$$\mathcal{S}_2(u, v) = \begin{cases} 2 & u = v \\ \infty & u = '-' \text{ \& } v = '-' \\ -1 & u \neq v, \text{ and } u, v \in \Sigma \\ -3 & \text{otherwise} \end{cases} \quad (10)$$

Now $\mathcal{S}_1(A1) = 6*2 + 3*(-3) + (-1) = 2$, while $\mathcal{S}_2(A2) = 5*2 + 1*(-3) + 3*(-1) = 4$. The problem of choosing the correct score function for a biological problem is important and actively researched, but we won't worry about it now. Instead, we focus our attention on computing the highest scoring alignment, given an appropriate score function.

Optimum alignment Consider sequences s and t over an alphabet Σ and a function $\mathcal{S} : \Sigma' \times \Sigma' \rightarrow \mathcal{R}$, where $\Sigma' = \Sigma \cup \{-\}$. Compute an alignment A^{opt} with the maximum score, defined by

$$\text{Score}(A^{opt}) = \max_A \{ \sum_j \mathcal{S}(A[1, j], A[2, j]) \} \quad (11)$$

An algorithm that scores every possible alignment is hopelessly inefficient (Why?). Can we do better with dynamic programming? Clearly, there is an objective function that must be optimized. In fact, we will focus for now on computing the *score* of

| | | | | | |
|---|---|---|---|---|---|
| A | G | A | T | - | - |
| A | - | A | T | T | A |

| | | | |
|---|---|---|---|
| C | C | A | G |
| C | C | T | G |

Figure 8: The alignment A_1 of strings AGATCCAG and AATTACCTG, arbitrarily broken into two. If A_1 is the highest scoring alignment, then the left part must be the highest scoring alignment of strings AGAT, and AATTA.

the optimum alignment, rather than the alignment itself. This is typical in dynamic programming problems. Recall from the *Warehouse-Capacity* problem that we computed the maximum needed capacity first, and subsequently retrieved the series of transactions that reached that capacity. Once we have an algorithm for computing the highest score, computing an alignment that actually achieves that score will turn out to be a matter of simply maintaining additional information.

We need to define sub-problems with a recursive structure. Given an alignment, break it off at some column. As an example, see Figure 8. The alignment A_1 aligning $s = \text{AGATCCAG}$ with $t = \text{AATTACCTG}$ (Figure 6) is broken at column 6. The left part is an alignment A' of the prefixes AGAT, and AATTA. The key observation is that A_1 is an optimal alignment only if A' is an optimum alignment of the prefixes. Of course, prior to computing the optimum alignment, we do not know which pair of prefixes is aligned. Therefore, we simply choose all possible pairs of prefixes from the two strings and compute their alignment. If s has length m , and t has length n , there are only mn distinct prefix pairs. Define $S(i, j)$ to be the highest score of an alignment of the prefix pairs $s[1 \dots i]$ and $t[1 \dots j]$. Clearly, it is sufficient to compute $S(i, j)$ for all $0 \leq i \leq m, 0 \leq j \leq n$ as $S[m, n]$ provides the desired answer. Consider the boundary cases first. If $j = 0$, then there is only one possible alignment which is the prefix $s[1 \dots i]$ being aligned with gaps. In this case, $S(i, j) = \sum_{k=1}^i \mathcal{S}(s[i], ' -')$. Likewise, if $i = 0$, then $S(i, j) = \sum_{k=1}^j \mathcal{S}(' -', t[j])$.

When $i > 0$ and $j > 0$, the discussion above already hints at the recursive structure of these sub-problems. We formalize this intuition below. Consider the right-most column of an optimal alignment of the prefixes $s[1 \dots i]$ and $t[1 \dots j]$. It must be one of three possibilities.

1. $s[i]$ is aligned to $t[j]$, and the columns to the left then have the highest scoring alignment for the prefixes $s[1 \dots i - 1]$, and $t[1 \dots j - 1]$.
2. $s[i]$ is aligned to a '- ', and the columns to the left contain the optimal alignments of the prefixes $s[1 \dots i - 1]$, and $t[1 \dots j]$.
3. A gap (' -') is aligned to $t[j]$, and the columns to the left contain the optimal alignments of the prefixes $s[1 \dots i]$, and $t[1 \dots j - 1]$.

As these are the only possibilities, the score of the optimal alignment of this prefix pair is simply the maximum of the scores from these possibilities.

$$S(i, j) = \begin{cases} \mathcal{S}(s[i], ' -') + S(i - 1, j) & \text{if } j = 0 \\ \mathcal{S}(' -', t[j]) + S(i, j - 1) & \text{if } i = 0 \\ \max \begin{cases} \mathcal{S}(s[i], t[j]) + S(i - 1, j - 1) \\ \mathcal{S}(s[i], ' -') + S(i - 1, j) \\ \mathcal{S}(' -', t[j]) + S(i, j - 1) \end{cases} & \text{if } i > 0, j > 0 \end{cases} \quad (12)$$

We had mentioned earlier that defining the sub-problems and their recursive structure was the difficult part of dynamic programming. It is relatively easy to translate the recursions into a dynamic programming algorithm for solving the problem. Procedure *Opt-Align()* (Figure ??), is an $O(nm)$ time dynamic programming algorithm to solve the *Optimal-Alignment* problem. We maintain a table $S[0 \dots m, 0 \dots n]$ to store the computed scores $S(i, j)$ of all prefix pairs. $S[m, n]$ contains the score of the optimum alignment. The entries $S[i, j]$ are filled in a bottom-up fashion so that no sub-problem is recomputed. Figure ?? displays the values of the table S for strings AGATCCAG and AATTACCTG, and the score function $calS_1$ (See Equation ??).

As in the case of the *Optimum Capacity Substring*, problem, we can compute not only the score of the optimal alignment, but the alignment itself. Additionally, we maintain another table, $C[1 \dots m, 1 \dots n]$ in which we store the 'choice' for each prefix pair. Thus if $S[i, j]$ is obtained by aligning s_i with t_j , $C[i, j] = \rightarrow$, implying that both i and j will be decremented in the previous column of the alignment. On the other hand, if $S[i, j] = \mathcal{S}(s[i], ' -') + S(i - 1, j)$, then $C[i, j] = \uparrow$ (only i is to be decremented). , and C for all i, j . The optimal alignment, obtained by tracing arrows starting from $C[m, n]$ is shown in Figure ??b.

The Edit-distance problem: Given two strings s , and t over an alphabet Σ (ex: $\Sigma = \{A, C, G, T\}$), compute the minimum number of edit operations (insertions, deletions and transformations) needed to transform s to t .

As stated, the edit-distance problem is a general problem in text comparison, with many applications. For example, it can be used as a test of plagiarism. In bioinformatics however, this is a key problem. Many of our genes code for proteins whose proper functioning is essential not only for humans but for all life-forms. **give an example here**. Indeed, a typical mouse ortholog of a human gene is almost 80% identical. Both sequences arose from a common ancestral sequence and diverged by accumulating mutations in the form of insertions, deletions and substitutions of nucleotides. Thus it is natural to ask for two homologous sequences (those that have arisen from a common ancestor), the edit-distance question so as to quantify the evolutionary distance between them.

A nice way to visualize the sequence of mutational events transforming one sequence into another is to give an *alignment*. As Figure 6 shows, scanning the alignment column by column gives a sequence of edits that transform sequence s into t . These are a deletion of G in column 2, followed by an insertion of A (column 5), and

finally a transformation of A into T (column 8). It is left to the interested reader to show that the score function below has the property that the alignment with the highest score is one that gives the minimum edit-distance between s and t .

4 Local Alignment: the Smith Waterman Algorithm

It is often the case in bio-molecular sequences that a sub-sequence of functional importance is embedded within two highly diverged sequences. As an example, consider the strings $X = \langle atatatataaaa \rangle$, and $Y = \langle aaaaaacacac \rangle$. The substring $aaaa$ is embedded in both X , and Y . The optimal global alignment of the two strings is shown in Figure ??a, and has a higher score than the alignment in Figure ??b, which aligns the embedded sequence correctly. In this case, we would like to see the best *local-alignment*, defined as highest scoring alignment of two substrings $s[i_1 \dots j_1]$ and $t[i_2 \dots j_2]$ over all such pairs of sub-strings. The naive approach to solving this would be to take all pairs of substrings ($O(n^2m^2)$ many), and compute a global alignment of each. Note however, that these problems share many sub-problems, and so we should be able to do better.

Define $S[i, j]$ as the score of the best local alignment that ends at s_i and t_j . As with global alignment, the right-most column of the alignment must be one of 3 cases, and therefore equation 12 must apply here as well. There is however, one important difference. Suppose every pair of substrings that end in s_i and t_j align with a negative score. Then, the best local alignment that ends in s_i and t_j is simply the null alignment, with score 0. Thus

$$S(i, j) = \max \begin{cases} \mathcal{S}(s[i], t[j]) + S(i-1, j-1) \\ \mathcal{S}(s[i], '-') + S(i-1, j) \\ \mathcal{S}('- ', t[j]) + S(i, j-1) \\ 0 \end{cases} \quad (13)$$

The choice array C is also modified to terminate whenever 0 is the highest scoring possibility.

$$C[i, j] = \begin{cases} \begin{matrix} ' \swarrow ' \\ ' \uparrow ' \end{matrix} & \text{if } S[i, j] = \mathcal{S}(s[i], t[j]) + S[i-1, j-1] \\ \begin{matrix} ' \uparrow ' \\ ' \leftarrow ' \end{matrix} & \text{if } S[i, j] = \mathcal{S}(s[i], '-') + S[i-1, j] \\ \begin{matrix} ' \leftarrow ' \\ ' ' \end{matrix} & \text{if } S[i, j] = \mathcal{S}('- ', t[j]) + S[i, j-1] \\ ' ' & \text{if } S[i, j] = 0 \end{cases} \quad (14)$$

Figure ?? displays the S , and C arrays for the strings $X = \langle atatatataaaa \rangle$, and $Y = \langle aaaaaacacac \rangle$. As desired, the highest scoring local alignment aligns the embedded string $\langle aaaa \rangle$.

5 Multiple Alignment

Figure ?? displays a *multiple-alignment* of XXXX sequences. One can view a multiple alignment as a generalization of the previously described *pairwise-alignment*.

However, multiple alignments have proved to be of great importance to molecular biology. For example, they highlight residues that are conserved over millions of years of evolutionary drift. As this is not a chance event, it could only be because the residue is important in maintaining the structure of the protein or has some other functional relevance. Also, in searching for distant homologs of a family, the sensitivity and specificity of search are improved considerably by scoring for consistency in conserved sites. For DNA sequences, multiple alignments are useful in sequence assembly, when many overlapping sequences must be aligned in the presence of sequencing errors to obtain the consensus sequence.

This said, computing good multiple alignments automatically is a problem of great interest, and has seen considerable action in the computational biology community. We discuss here a generalization of the pairwise-alignment score function that is amenable to a dynamic programming approach. Consider n sequences S^1, S^2, \dots, S^n over an alphabet Σ of lengths l_1, l_2, \dots, l_n respectively. Define a multiple alignment M of n sequences by a matrix of n rows, and m columns. Each row i corresponds to the sequence S^i with some interspersed gaps. Correspondingly, each column of M corresponds either to a symbol or to a gap '-', but every column must have at least one non-gap symbol. Each column j is defined as a vector $\vec{j} \in (\Sigma \cup \{-'\})^n$ and assigned a score $\mathcal{S}[\vec{j}]$. The score of an alignment is sum of scores of all columns vectors. The *Optimal-Multiple-Alignment* problem is now the problem of computing a multiple alignment with the highest score.

We can solve the Optimal-Multiple-Alignment problem by considering optimal alignments of all prefixes of all strings. Each set of prefixes can be defined by choosing an appropriate $\vec{p} = \langle p_1, p_2, \dots, p_n \rangle$, with $1 \leq p_j \leq l_j$ for all j . Now consider the rightmost column of any alignment of the prefix set \vec{p} . The set of feasible columns $\mathcal{C}_{\vec{p}}$ is defined by

$$\mathcal{C}_{\vec{p}} = \{\vec{s} : s_j \in \{S^j[p_j], '-'\} \forall j\} \quad (15)$$

Also define the preceding prefix of \vec{p} corresponding to a column $\vec{s} \in \mathcal{C}_{\vec{p}}$ by the vector $\vec{q}^{\vec{p}, \vec{s}}$, where

$$q_i^{\vec{p}, \vec{s}} = \begin{cases} p_i - 1 & \text{if } s_i = S^i[p_i] \\ p_i & \text{if } s_i = '-'$$

Then the score of a prefix $S[\vec{p}]$ can be defined as

$$S[\vec{p}] = \max_{\vec{s} \in \mathcal{C}_{\vec{p}}} \{S[\vec{q}^{\vec{p}, \vec{s}}] + \mathcal{S}[\vec{s}]\} \quad (16)$$

6 RNA Secondary Structure

RNA is one of the three major classes of bio-molecules, the other two being DNA and Protein. As the name suggests, RNA is composed of nucleic acids, much like DNA, but with Thymine ('T') replaced by Uracil ('U'). For our purposes, it is often sufficient to think of RNA as a string over a 4 letter alphabet, $\Sigma = \{A, C, G, U\}$. **(NOTE: DNA has not been defined, nor has an introduction been given to basic Biology).** The nucleotides are reactive **Check a biochem. textbook,**

and tend to pair with other nucleotides by forming hydrogen bonds, typically using the *Watson-Crick* pairing, $A \leftrightarrow T$, and $C \leftrightarrow G$. This leads to another major difference between RNA and DNA. DNA is double stranded and is stabilized by two complementary strands pairing with each other and forming a double helix. On the other hand, RNA is single stranded, and its nucleotides pair with complementary nucleotides on the same strand. A stacking of these interactions contributes to thermodynamic stability, and formation of local double-stranded regions. A collection of these base-pair interactions defines a *secondary structure* for the RNA molecule. Dynamic programming plays a key role in computing energetically favorable RNA secondary structures. To illustrate, we work with a simplified models of RNA secondary structure and the energetics of those structures. Consider an RNA sequence t with $|t| = n$. The secondary structure is described by a set S containing all index pairs (i, j) , $1 \leq i < j \leq n$, such that t_i and t_j are complementary and form a base-pair. We will consider secondary structures that are free of *pseudo-knots*. In other words for all $(i_1, j_1), (i_2, j_2) \in S$, either $i_1 \leq i_2 < j_2 \leq j_1$, or $i_2 \leq i_1 < j_1 \leq j_2$. Also, based on the assumption that a structure with a larger number of base-pairs is more stable than one with fewer base-pairs, we define the *score* of a secondary structure S simply as the number of base-pairs in S . The *Optimal-RNA-Secondary-Structure* problem can be formulated as follows:

Optimal-RNA-Secondary-Structure: Given an RNA string t of length n , compute a pseudo-knot free secondary structure S with maximum number of base-pairs.

What is the recursive sub-structure in this problem? Clearly, it is sufficient to compute the optimal secondary structure $E(i, j)$ for every substring $t[i \dots j]$, with $1 \leq i < j \leq n$. To identify if these problems have a recursive sub-structure, we consider all of the ways in which $t[i]$ and $t[j]$ can form base-pairs (or not). As in the case of alignment, there are only a few possibilities:

1. $t[i]$ does not form a base-pair. In this case, $E(i, j) = E(i + 1, j)$.
2. $t[j]$ does not form a base-pair. In this case, $E(i, j) = E(i, j - 1)$.
3. $t[i]$ and $t[j]$ are complementary, and pair with each other. Consequently, $E(i, j) = 1 + E(i + 1, j - 1)$.
4. There exists $i < k < j$, such that $t[k]$, and $t[j]$ are complementary, and pair up. Then, all base-pairs (l, m) in a secondary structure of $t[i \dots j]$ must have the property that either $i \leq l < m < k$, or $k < l < m < j$. Otherwise, (l, m) and (k, j) would form a pseudo-knot. In this case, then $E(i, j) = 1 + E(i, k - 1) + E(k + 1, j - 1)$.

As these are the only possibilities, the energy of the optimal secondary structure for $t[i \dots j]$ is obtained by taking the maximum.

$$E(i, j) = \max \begin{cases} E(i+1, j) \\ E(i, j-1) \\ 1 + E(i+1, j-1) \\ \max\{1 + E(i, k-1) + E(k+1, j-1)\} \end{cases} \quad \begin{array}{l} \text{if } t[i] \leftrightarrow t[j] \\ \forall i < k < j \text{ s.t. } t[k] \leftrightarrow t[j] \end{array} \quad (17)$$