Genetic Subsequencing Algorithm

Pamela Bilo

Alina Banerjee

B649 – Fall 2012

December 5, 2012

What are we trying to accomplish?

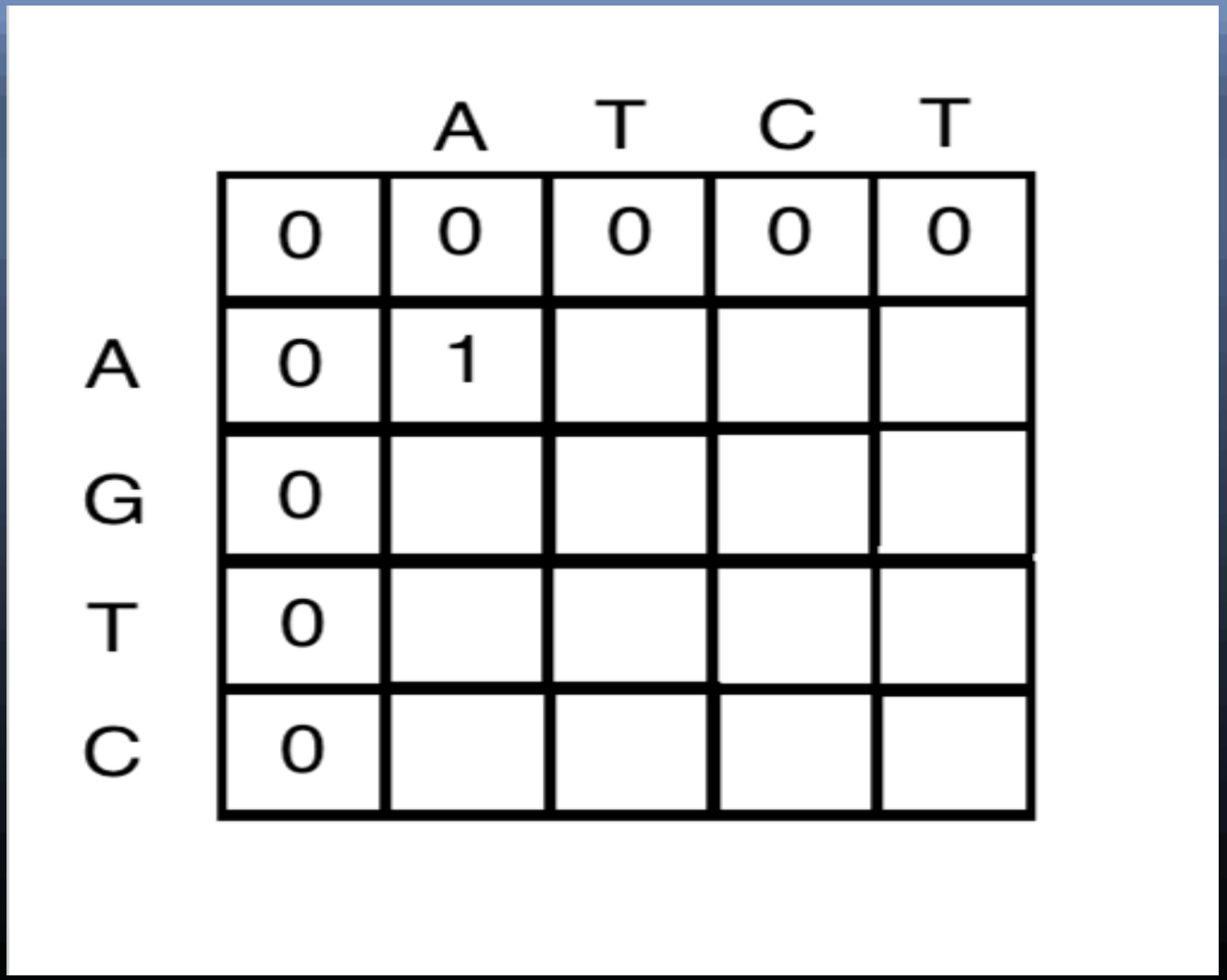
- Genetic subsequencing – finding similarities between strands of DNA
- DNA consists of four bases: guanine (G), adenine (A), thymine (T), and cytosine (C). A DNA strand is a combination of these four bases of length 3 billion
- It can be determined how closely related two organisms are by comparing their longest common subsequence.

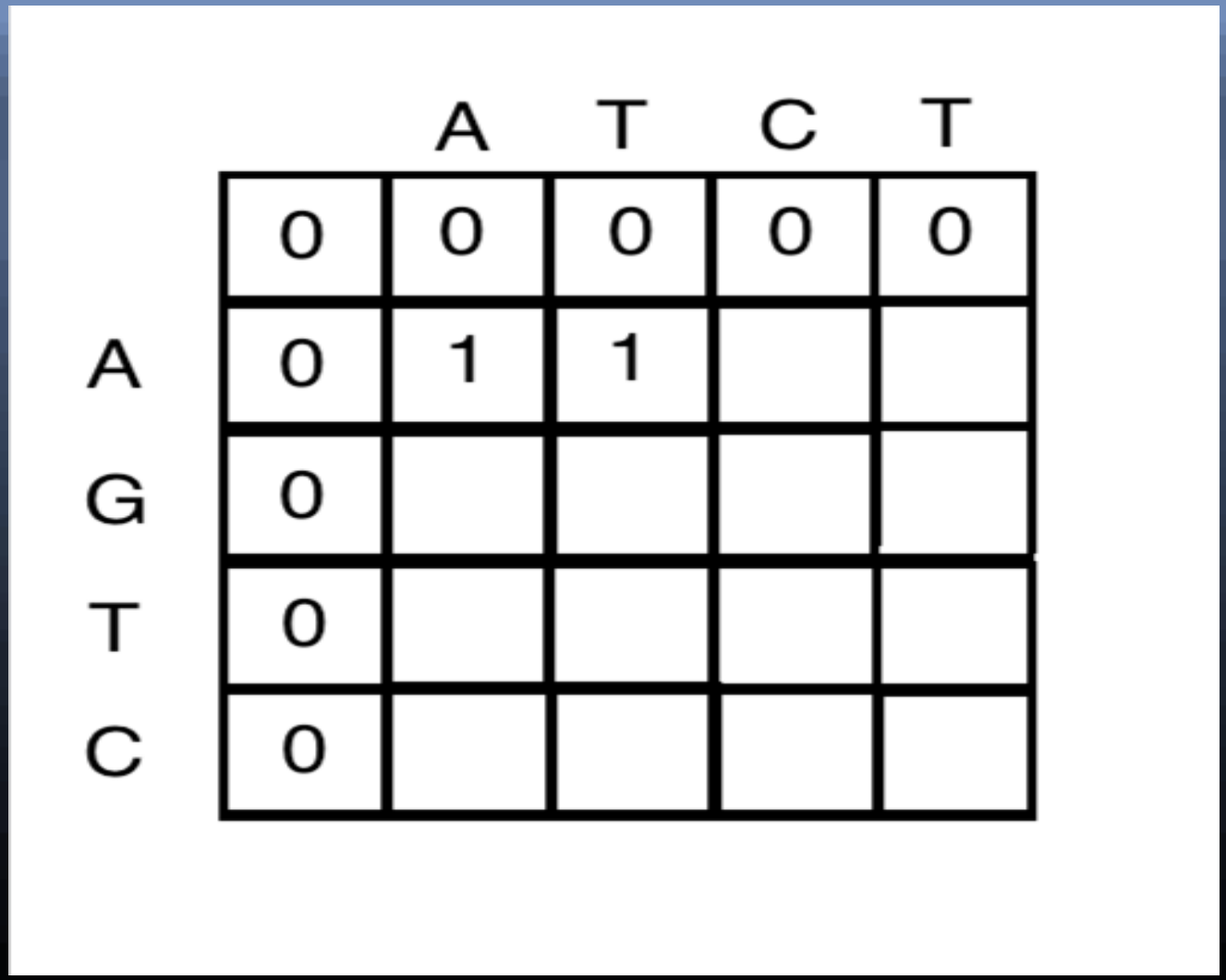
What is a subsequence?

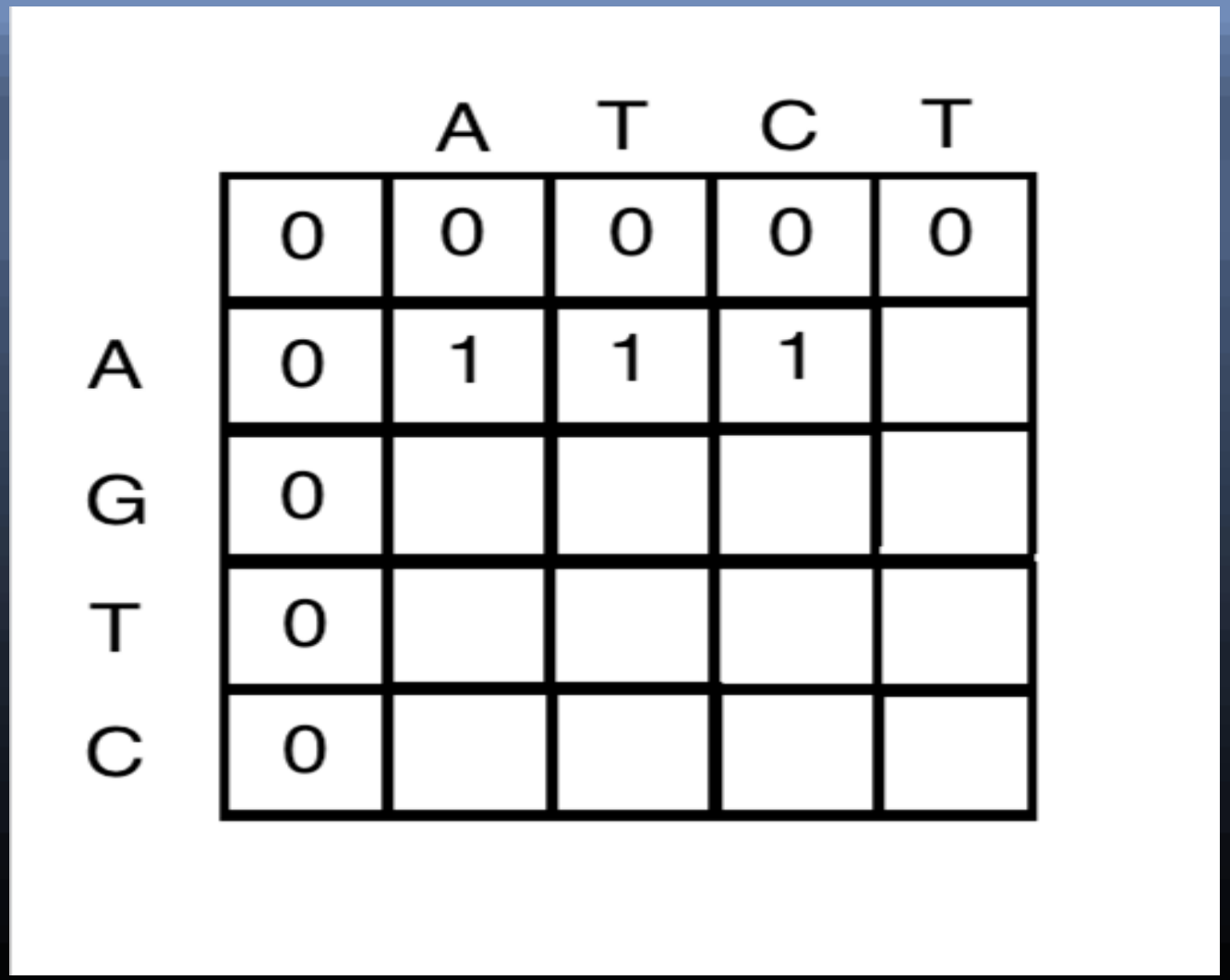
- A subsequence of a sequence A = (a0, a1, a2, a3, ... an) is any combination of ai such that the index at position i is less than that of position i + 1.
- Example: In the sequence “ABCDEF”, examples of subsequences would be “ABC”, “ABD”, “ACF”. “FCA” would not be a subsequence because F appears after C in the original sequence
- The longest common subsequence problem is the task of finding the longest subsequence that belongs to sequence A and sequence B.

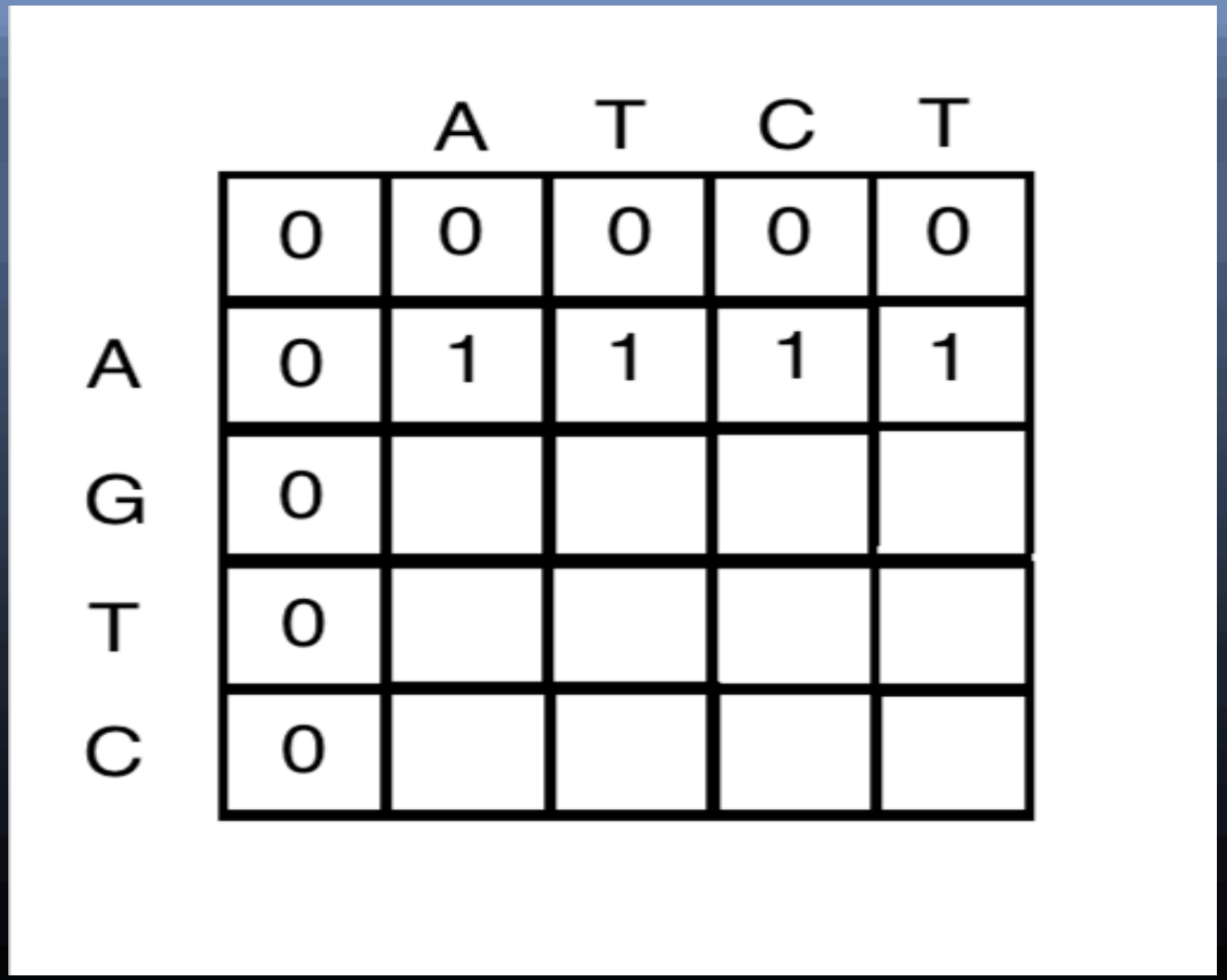
The Algorithm

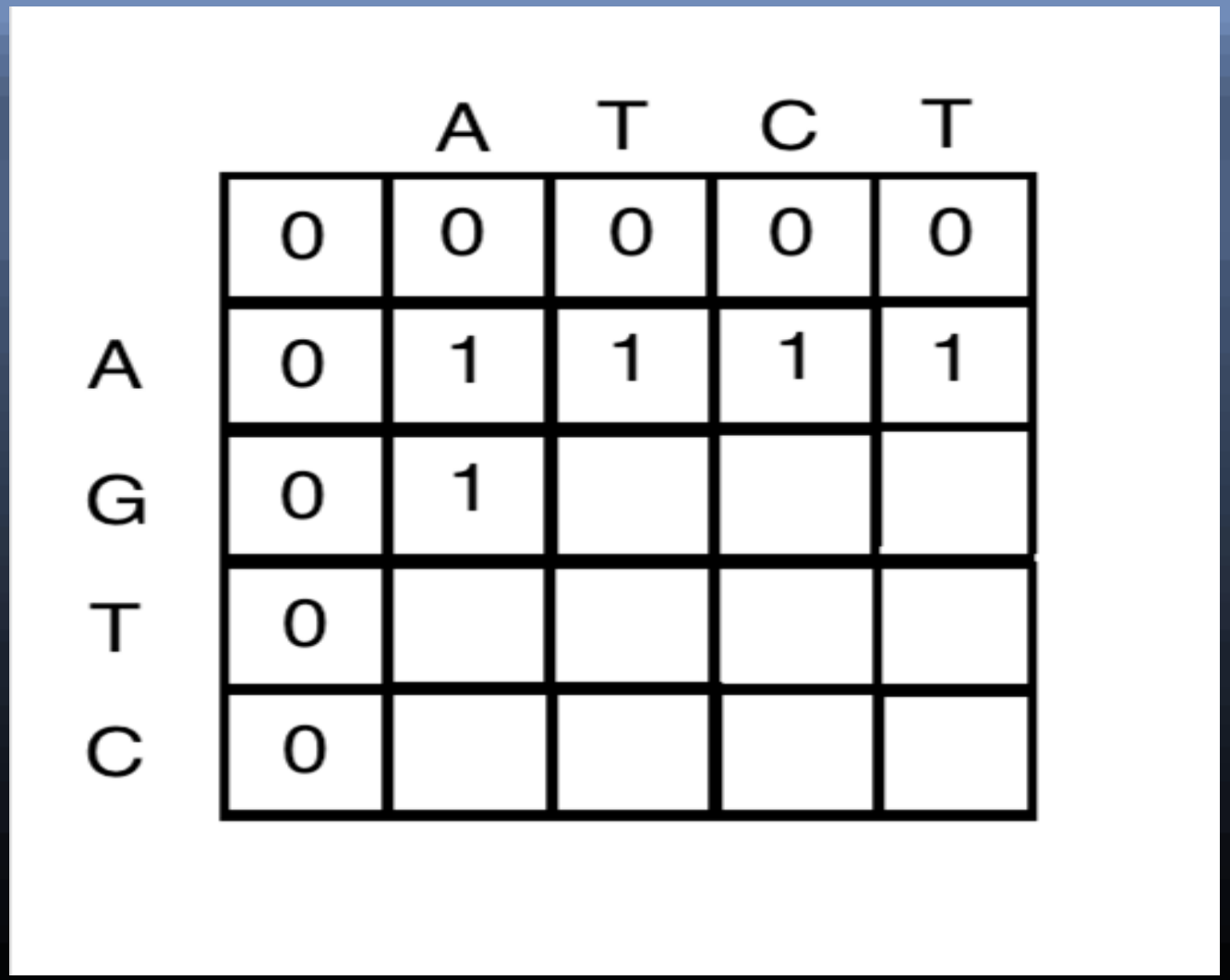
- Construct a matrix of size (i + 1) x (j + 1), where i and j are the lengths of A and B, respectively.
  - Fill row 1 and column 1 with zeroes
  - Traverse down the matrix. For each position (i, j) in the matrix:
    - If a[i] = b[j], m(i, j) = m(i – 1, j – 1)
    - Otherwise, m(i, j) = max(m(i – 1, j), m(i, j - 1))

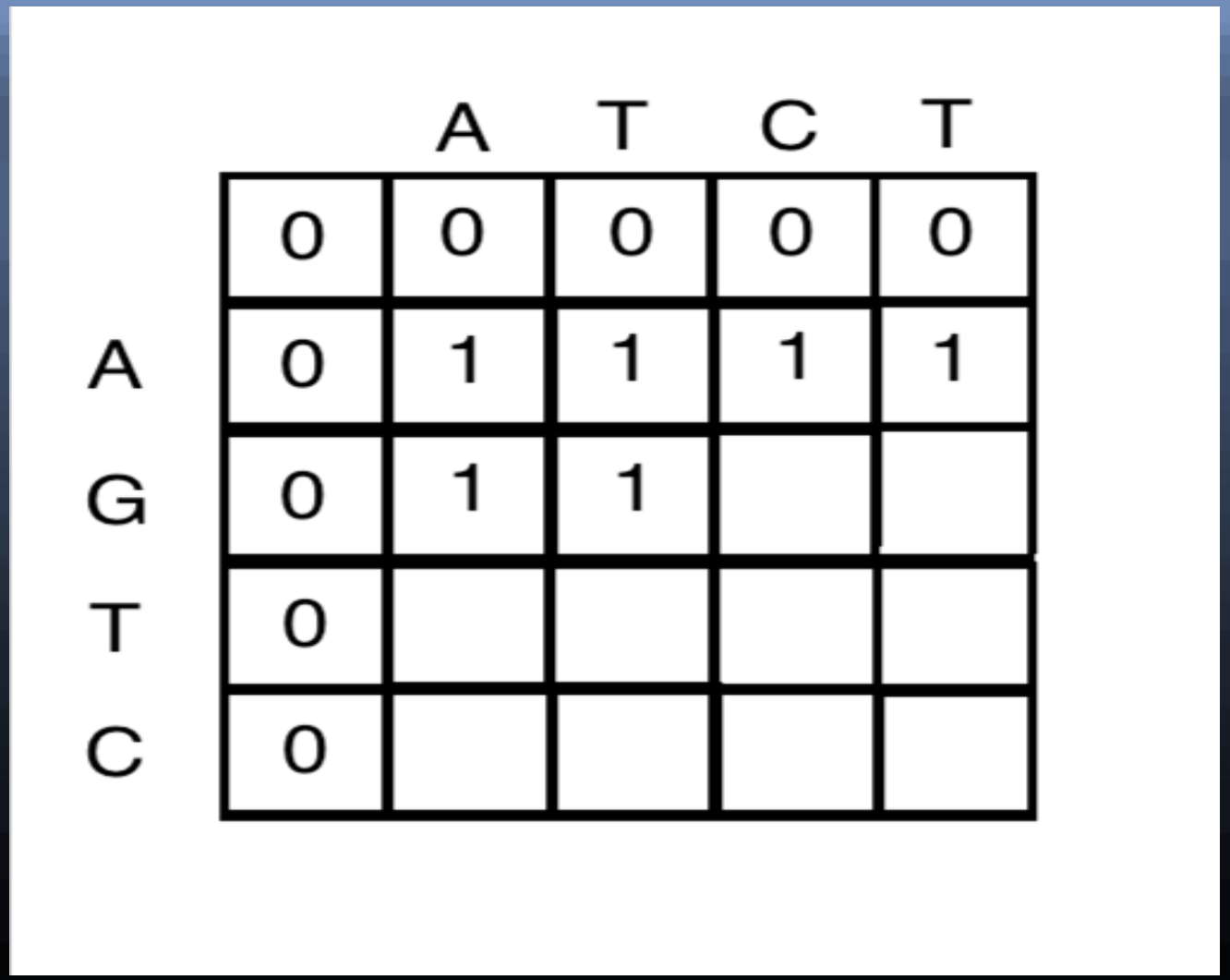

A T C T
0 0 0 0 0
A 0 1
G 0
T 0
C 0

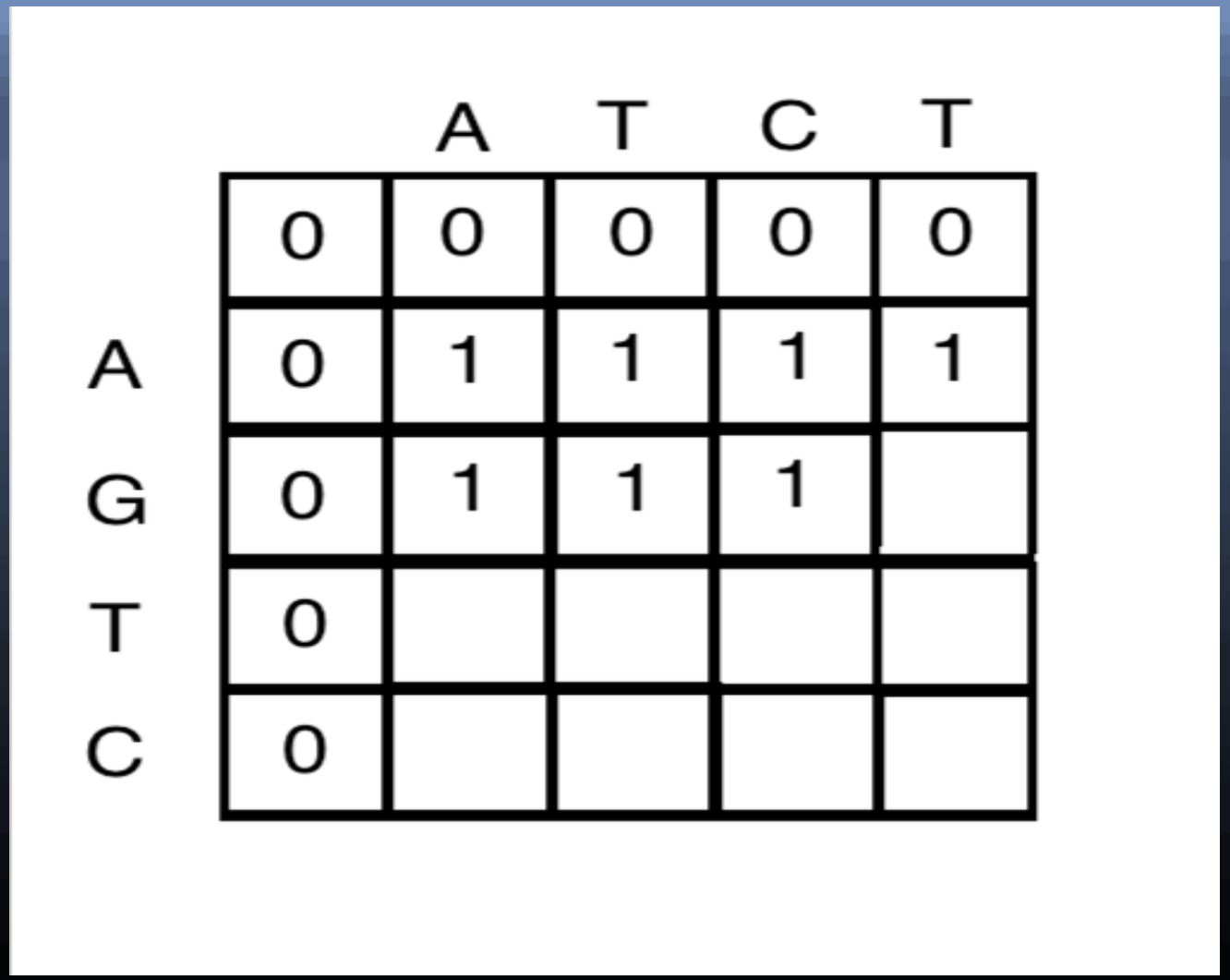

A T C T
0 0 0 0 0
A 0 1 1
G 0
T 0
C 0

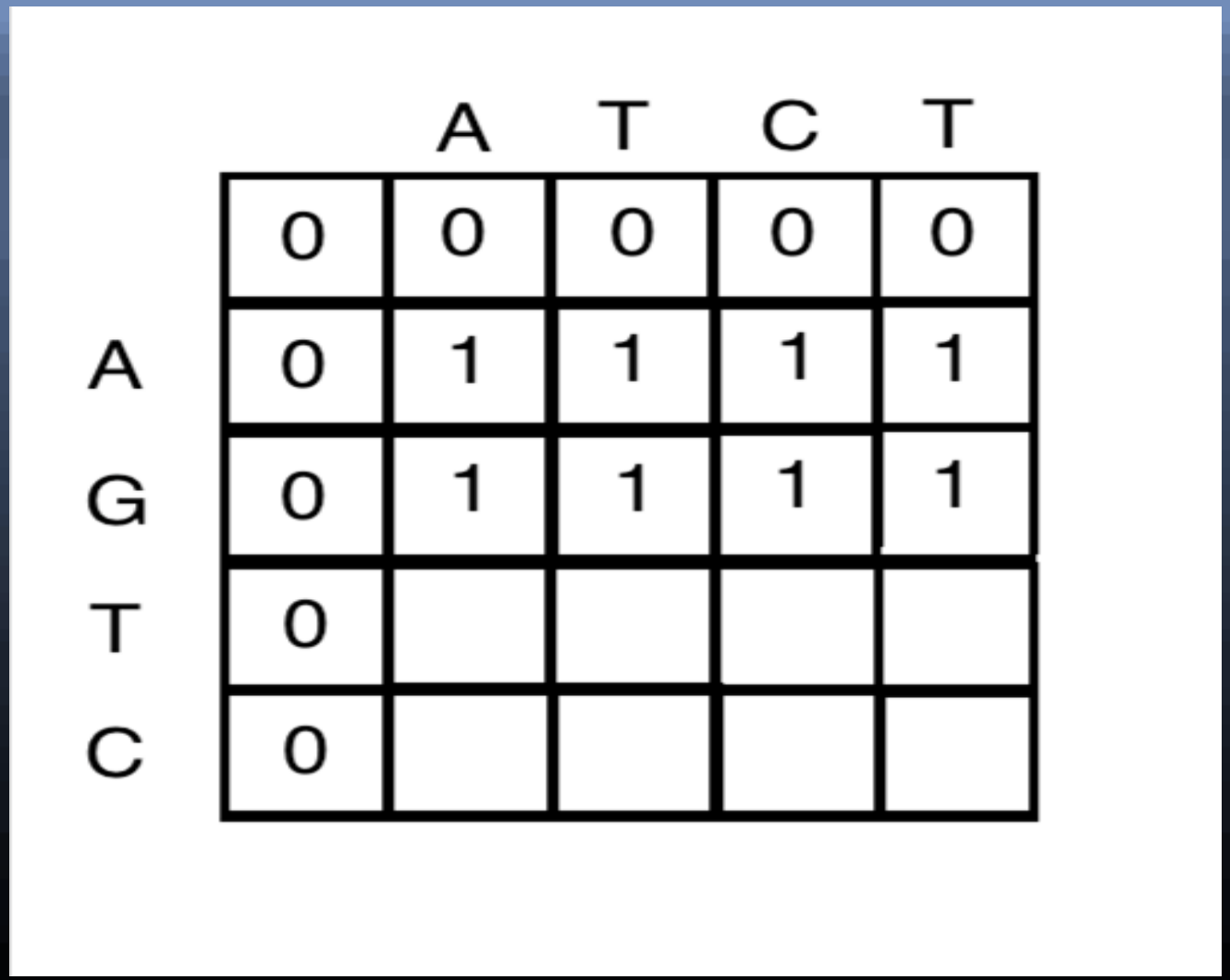

A T C T
0 0 0 0 0
A 0 1 1 1
G 0
T 0
C 0

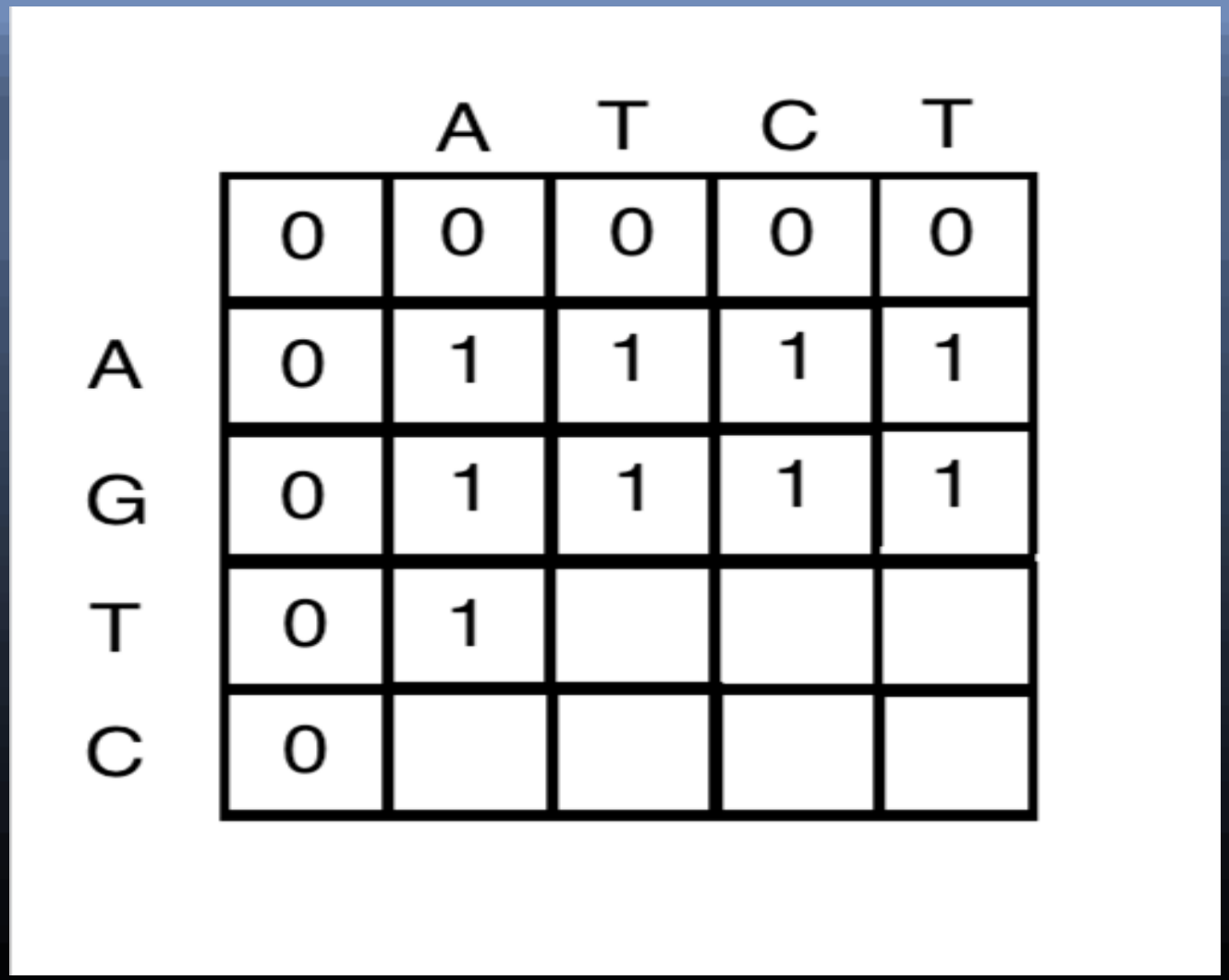

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0
T 0
C 0

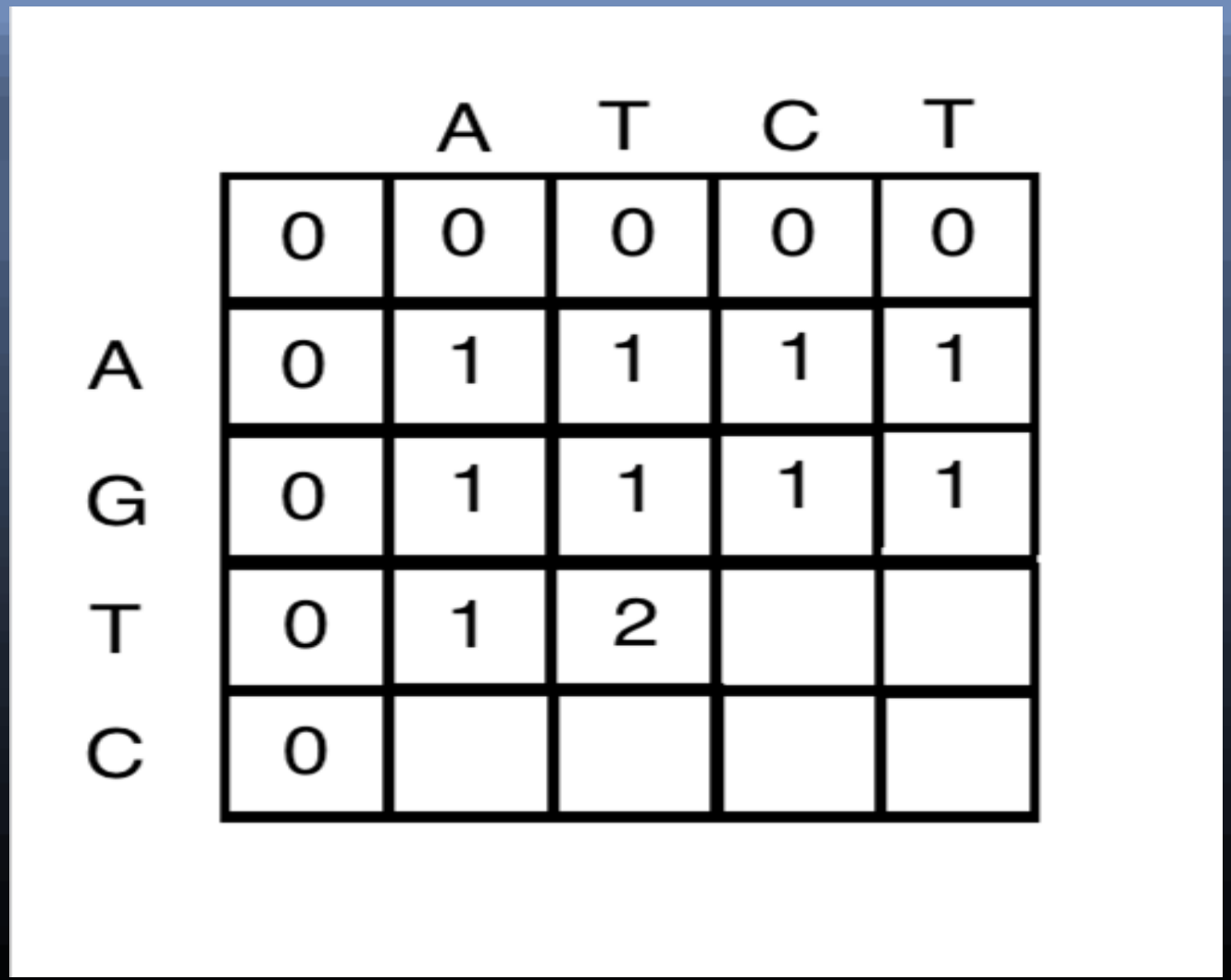

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1
T 0
C 0

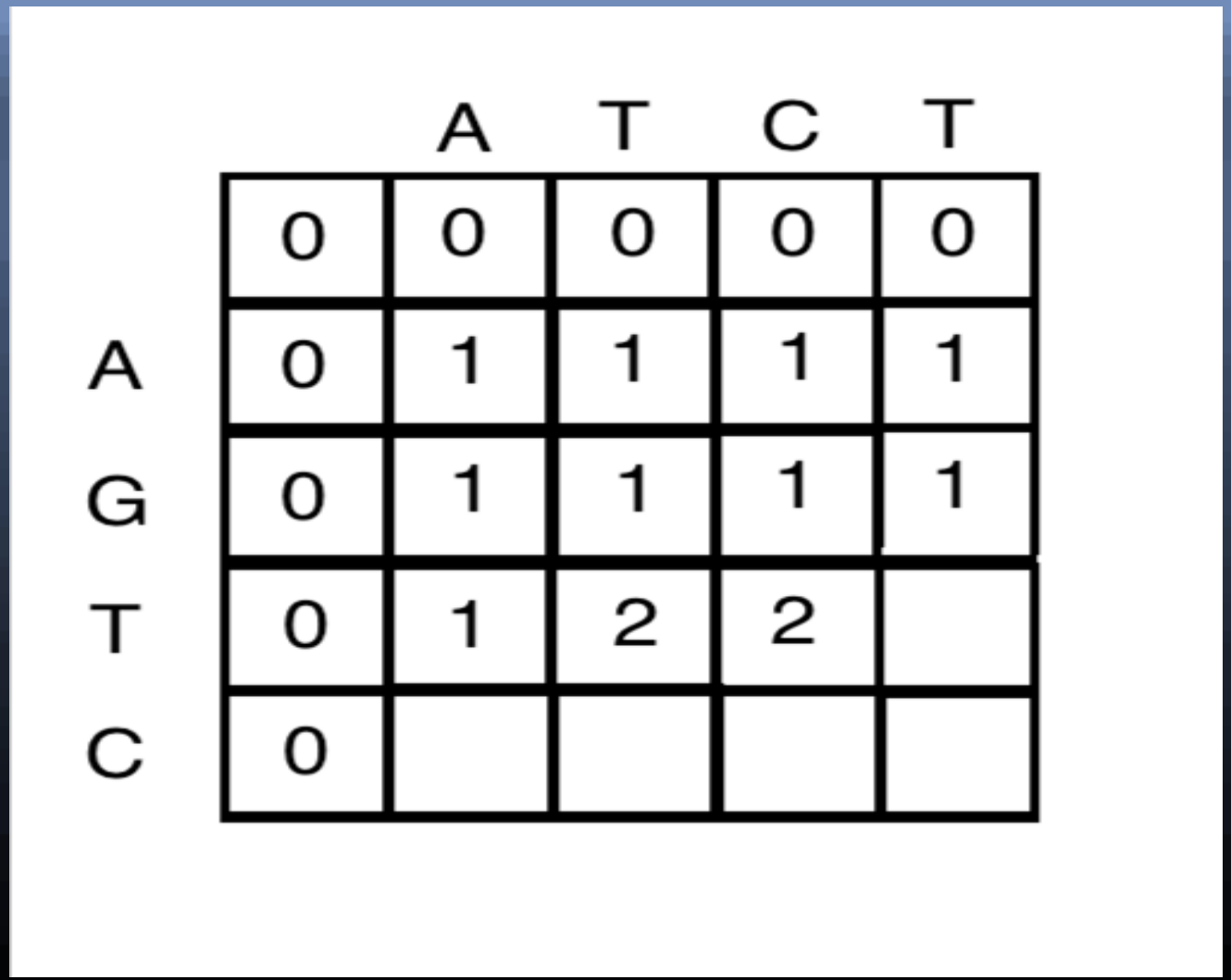

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1
T 0
C 0

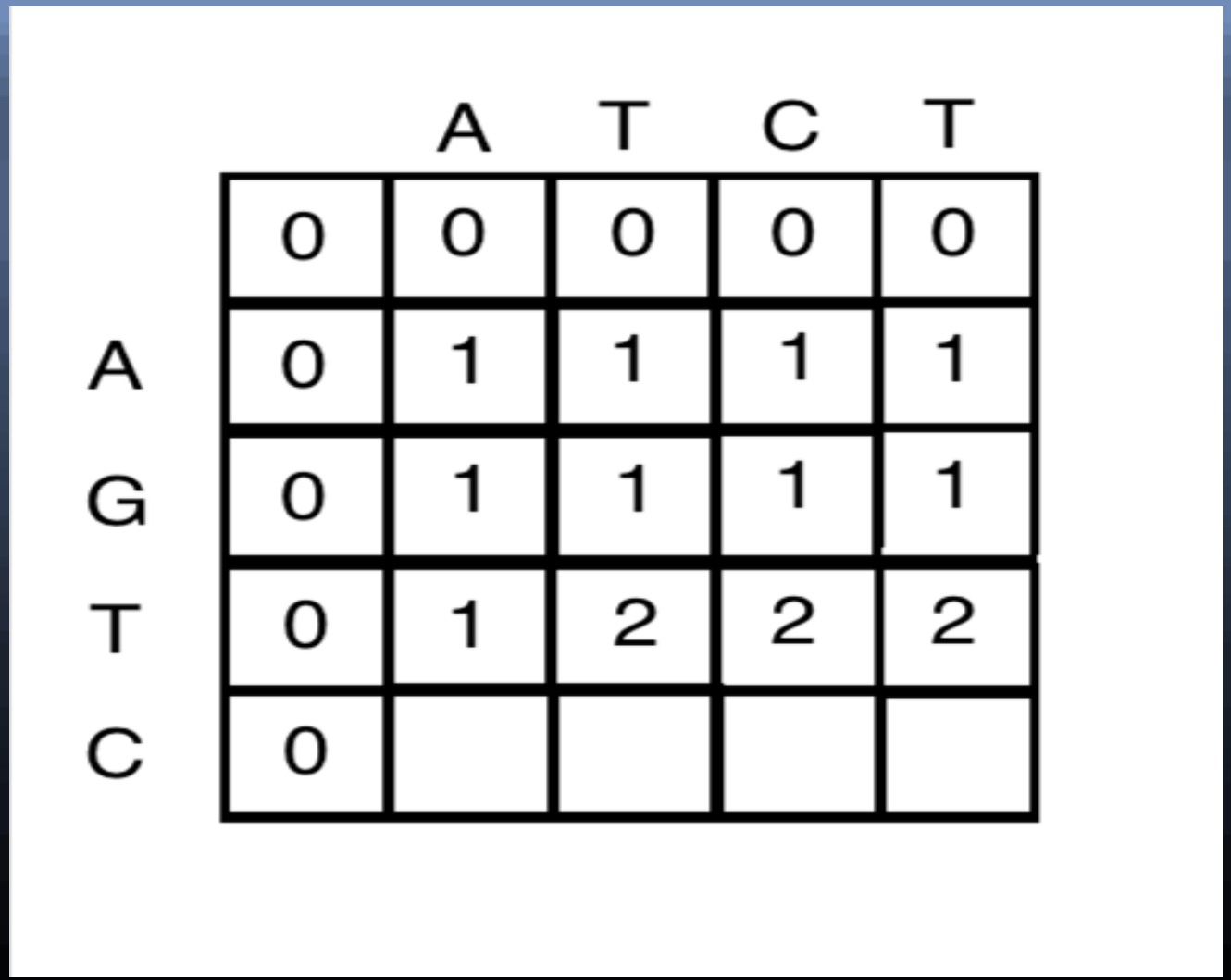

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1
T 0
C 0

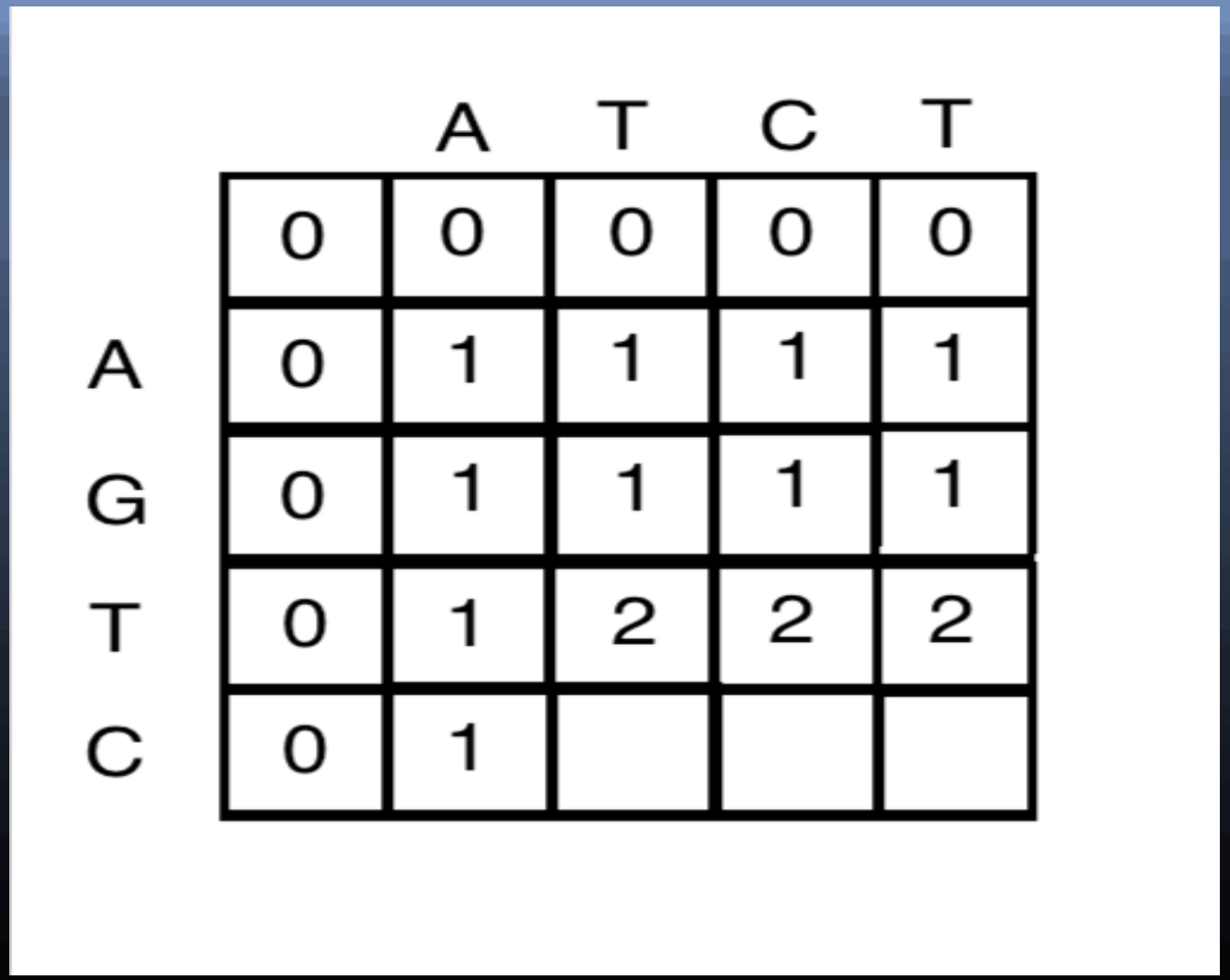

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0
C 0


A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1
C 0


A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1 2
C 0


A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1 2 2
C 0

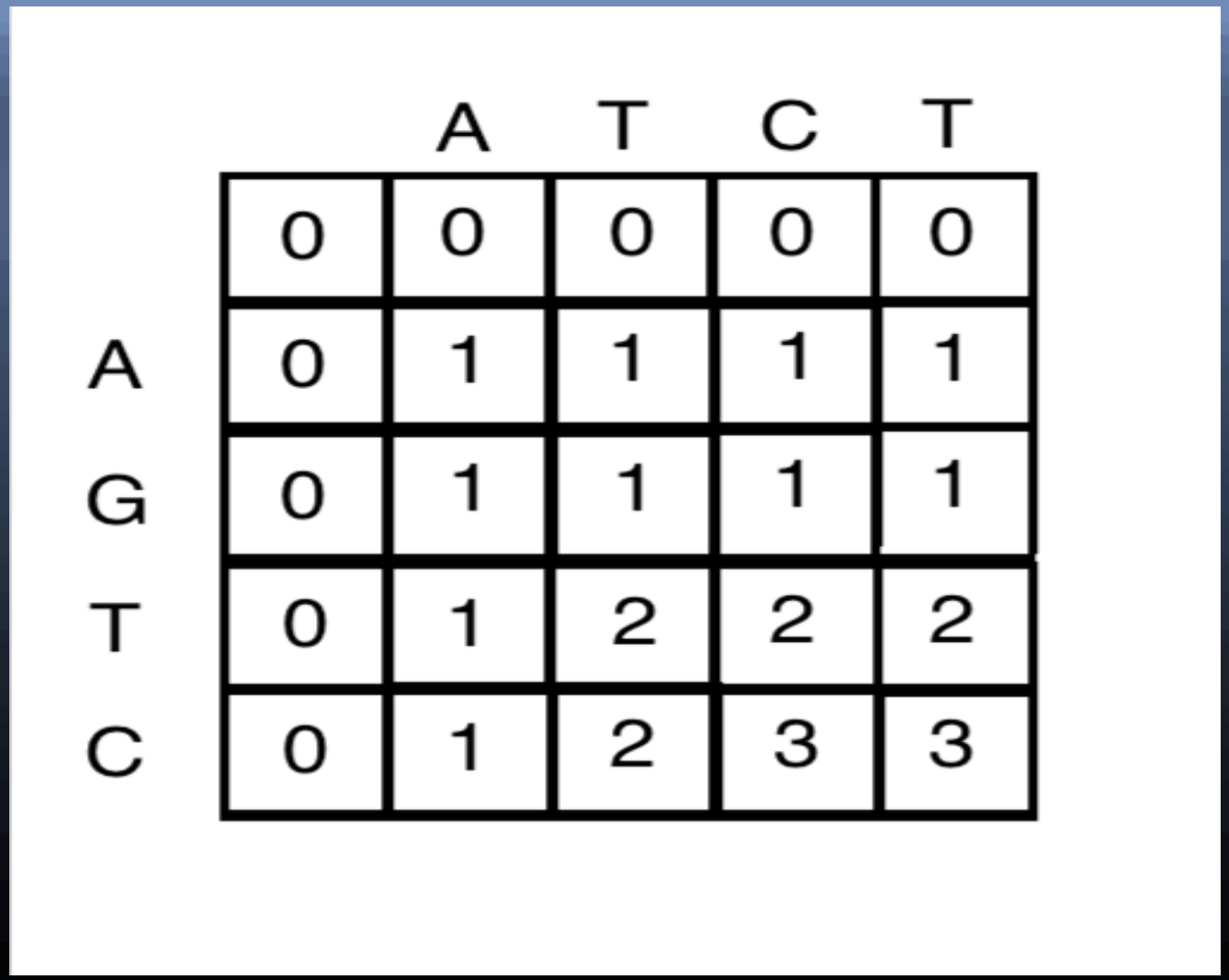

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1 2 2 2
C 0


A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1 2 2 2
C 0 1

		A	T	C	T
	0	0	0	0	0
A	0	1	1	1	1
G	0	1	1	1	1
T	0	1	2	2	2
C	0	1	2		

		A	T	C	T
	0	0	0	0	0
A	0	1	1	1	1
G	0	1	1	1	1
T	0	1	2	2	2
C	0	1	2	3	

		A	T	C	T
	0	0	0	0	0
A	0	1	1	1	1
G	0	1	1	1	1
T	0	1	2	2	2
C	0	1	2	3	

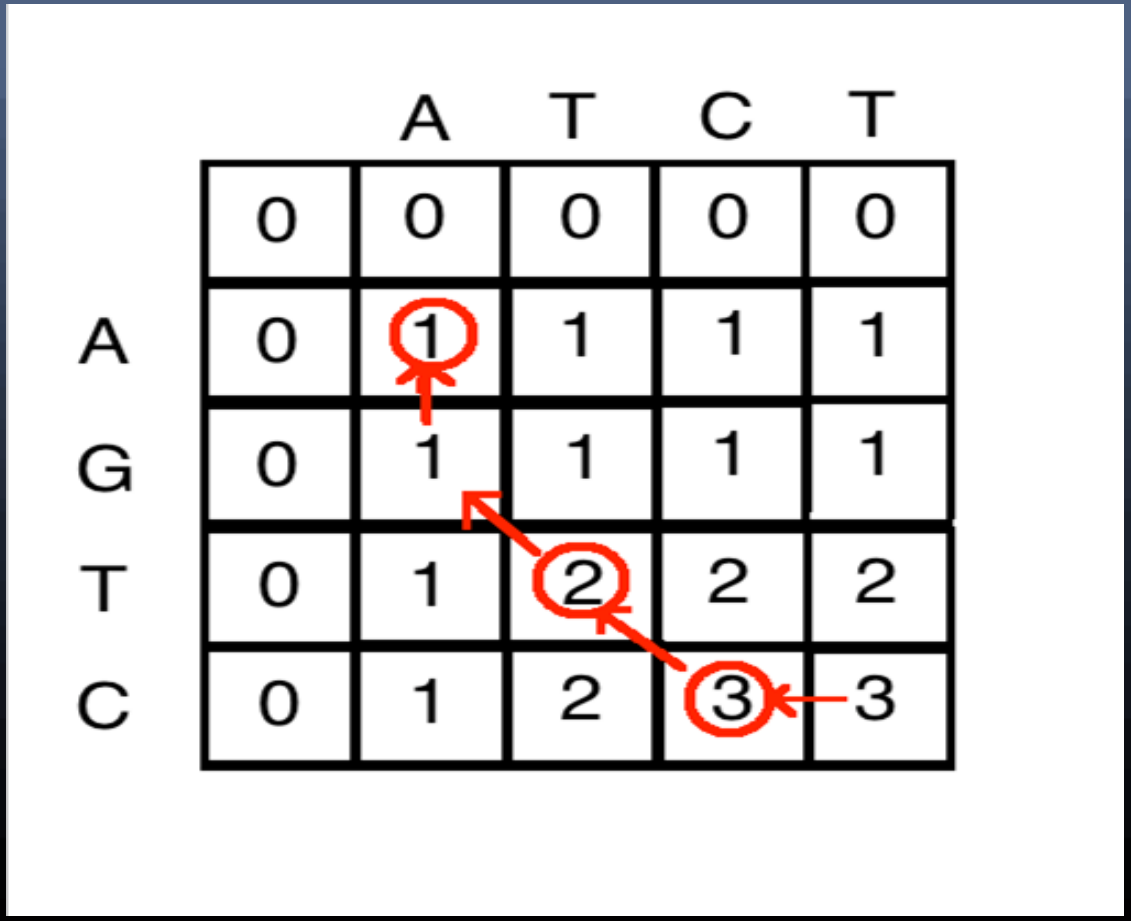

A T C T
0 0 0 0 0
A 0 1 1 1 1
G 0 1 1 1 1
T 0 1 2 2 2
C 0 1 2 3 3

Subsequence Construction
~~Backtracking~~

- To find the longest common subsequence, start at the bottom right corner.
  - If a[i] != b[j]:
    - If m(i – 1, j) > m(i, j -1):
      - Move to position m(i – 1, j)
      - Else move to position m(i, j – 1)
  - Else:
    - Add a[i] to the longest common subsequence
    - Move to position m(i – 1, j - 1)

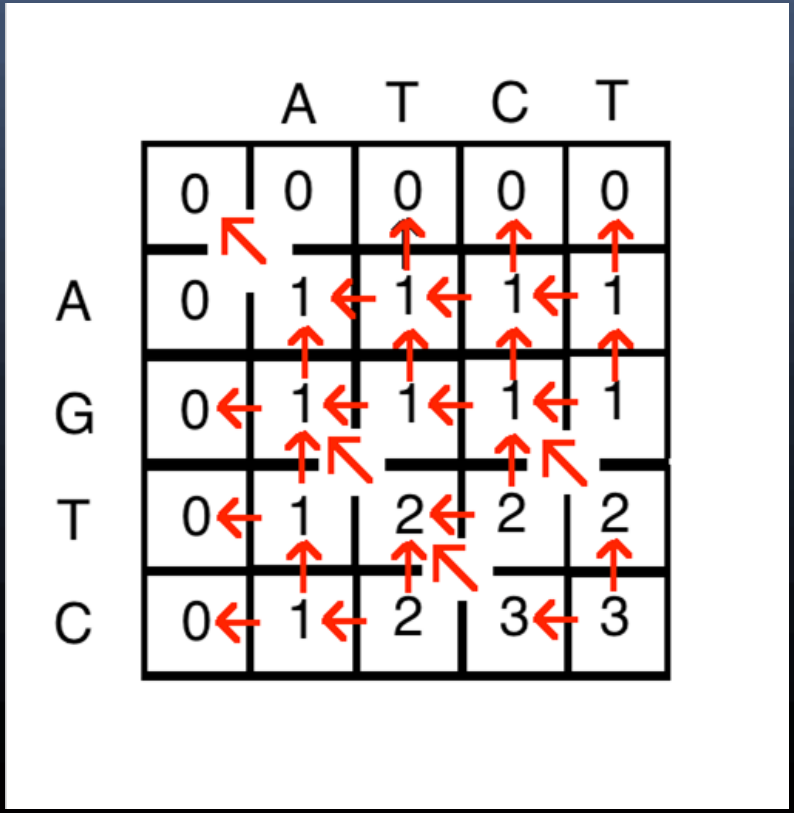
Subsequence Construction
~~Backtracking~~

- In this case, the longest common substring is "ATC". We can look at our original strings and see that "ATCT" and "AGCT".



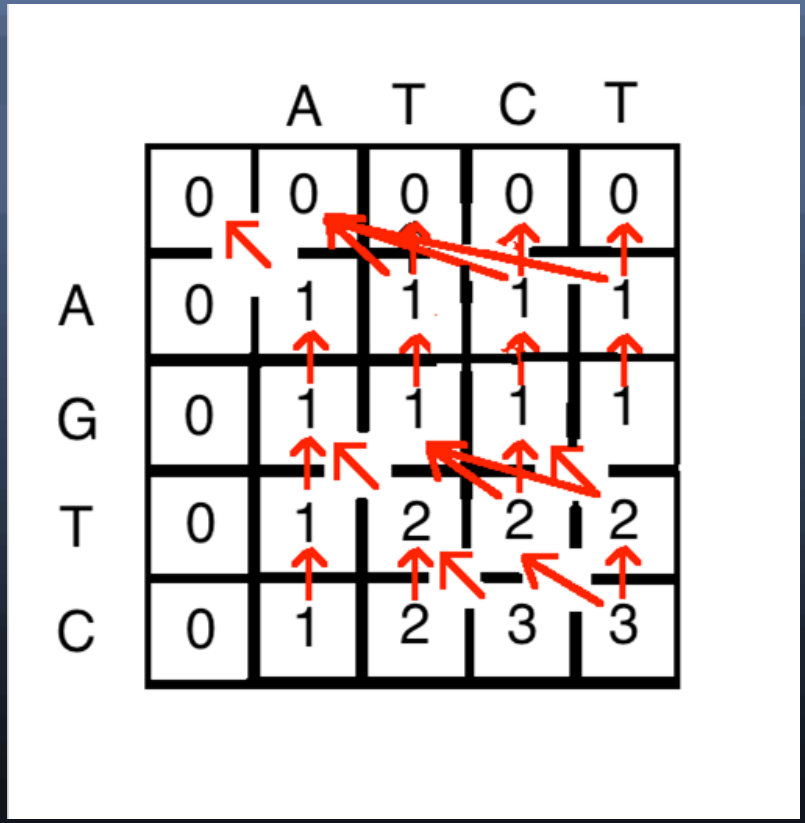
Analysis of serial algorithm

- For every m(i, j) in the matrix, there are three dependencies: between m(i – 1, j), m(i – 1, j – 1), and m(i, j – 1). This is impossible to parallelize.



Changing dependencies

- We want to manipulate the algorithm such that ALL dependencies are on the previous row. We scan the previous row for the last time the number appeared, and add one to it. Then, we find the maximum value of that number, and the number above it.

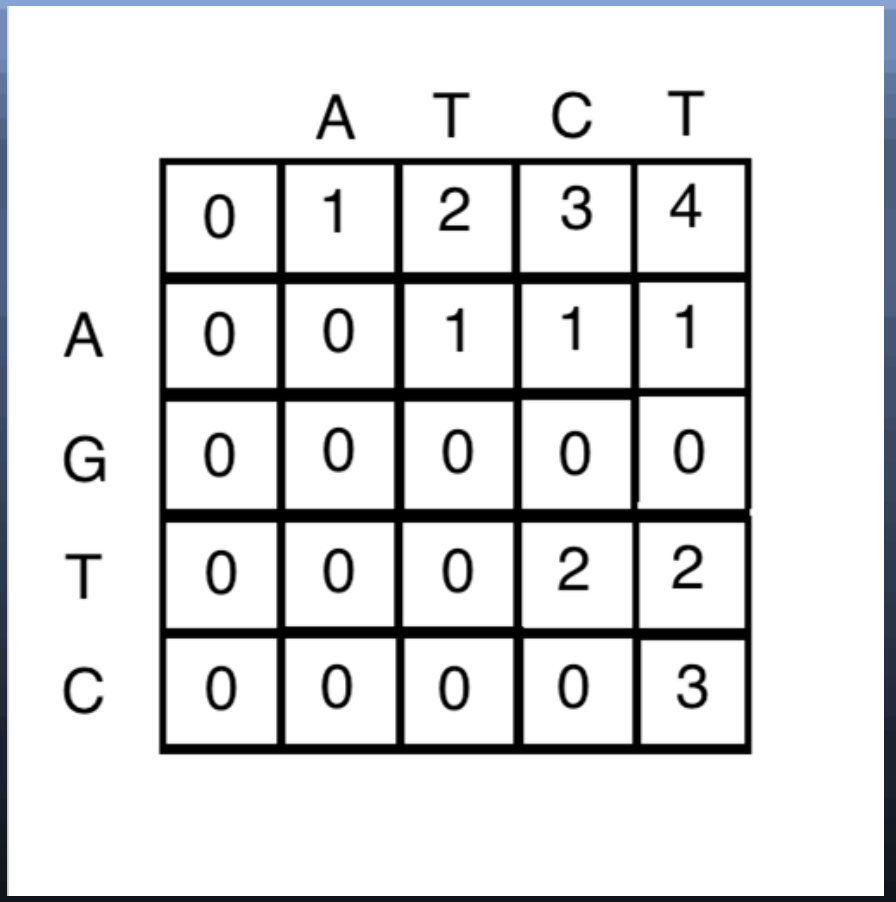


Precomputation Matrix

- Scanning our rows is a potentially time-consuming operation.
- We want to construct a matrix beforehand that will tell us exactly what column the number that we want is in.
- We will be able to directly access that information based on our position in the matrix.

Precomputation Matrix

- There are n + 1 rows where n is the number of characters that make up our sequence. Each row corresponds to that character. There are m columns where m is the number of elements in sequence A
- Position (0, j) is equal to j
- Position (i, 0) is equal to 0
- Else:
  - If (c[i] = a[j - 1]):
    - m(i, j) = j – 1
- Else:
  - m(i, j) = m(i, j - 1)

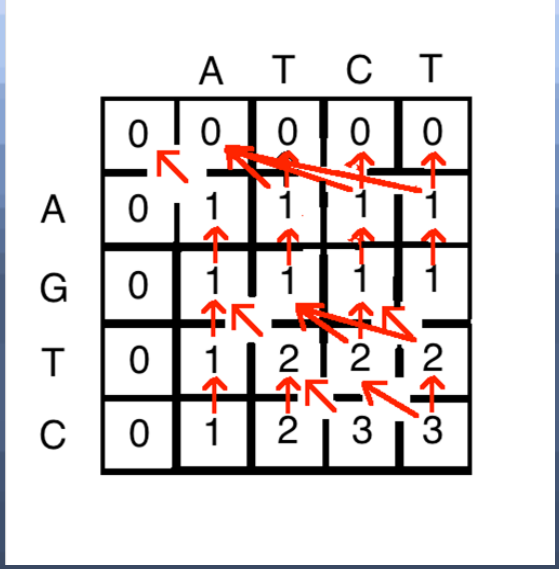


		A	T	C	T
	0	1	2	3	4
A	0	0	1	1	1
G	0	0	0	0	0
T	0	0	0	2	2
C	0	0	0	0	3

Putting it all together

- Now, we can look up where we need to go to in the previous row by accessing our precomputation matrix! Each entry in the precomputation matrix corresponds to the column in which the character last appeared.
- We can now slightly modify our algorithm.

New efficient parallelization



		A	T	C	T
	0	1	2	3	4
A	0	0	1	1	1
G	0	0	0	0	0
T	0	0	0	2	2
C	0	0	0	0	3

- If a[i] = b[j],
  - m(i, j) = m(i – 1, j – 1)
- Else if (p(c, j) = 0),
  - m(i, j) = m(i – 1, j)
- Else max(m(i – 1, j),
- m(i – 1, p(c, j) – 1) + 1)

P is our precomputation matrix. Coincidentally, the sequences on the row and column axis of both matrices are the same. This will not only be the case, for the values for each row in the precomputation matrix only consist of the letters that make up the sequence (A, G, T, C). However, this example makes it easier to understand

C is the number in which the character b[j] appears in the rows of P. In this case, c will always equal j. All b[j]'s where b[j] = A will map to c = 1, b[j]'s where bj = G maps to c = 2, etc.

P(c, j) tells us the column where the last instance of letter a[i] occurred. We use this to look to the previous row to find a value for m(i, j).

Potential Improvements

- Moving large matrix m to the GPU in blocks
- Using a binary matrix where m(i, j) = 0 if a[i] != b[j] and m(i, j) = 1 otherwise, and build down by assigning m(i, j) = m(i, j) + max(m(i – 1, j), m(i, j - 1))

References

- An Efficient Parallel Algorithm for Longest Common Subsequence Problem on GPUs. Jiaoyun Yang, Yun Xu, Yi Shang. 2010