```
"Begin digression: Model base pairing with a simple Clojure map."

(str "Model" \space "base" \space "pairing" \.) ; => "Model base pairing."

(def sentence ["Model" \space "base" \space "pairing" \.])
sentence ; => ["Model" \space "base" \space "pairing" \.]

(apply str sentence)                            ; => "Model base pairing."
((partial apply str) sentence)                  ; => "Model base pairing."

(def string (partial apply str))
(string sentence)                               ; => "Model base pairing."

nucleotides            ; => ["Adenine" "Cytosine" "Guanine" "Thymine"]

(first nucleotides)                     ; => "Adenine"
(first (second nucleotides))            ; => \C
(map first nucleotides)                 ; => (\A \C \G \T)

(def ACGT (string (map first nucleotides)))
ACGT                                    ; => "ACGT"
(string (reverse ACGT))                 ; => "TGCA"

(def pair (zipmap ACGT (reverse ACGT)))
pair                                    ; => {\A \T \C \G \G \C \T \A}

(map pair ACGT)                         ; => (\T \G \C \A)

[(rand-nth ACGT) (rand-nth ACGT)]       ; => [\G \A]

(def strand (repeatedly (fn [] (rand-nth ACGT))))
(string (take 23 strand))               ; => "TCTGTCCCCGTAGACAAGACGTT"
(string (take 23 (map pair strand)))    ; => "AGACAGGGGCATCTGTTCTGCAA"

"End digression: Now where were we?" "We were counting things."

(count ACGT)                                    ; =>  4
(count amino-acids)                             ; => 20
(count (for [x ACGT]                [x]))       ; =>  4
(count (for [x ACGT y ACGT]         [x y]))     ; => 16
(count (for [x ACGT y ACGT z ACGT] [x y z]))    ; => 64
```

Overlapping code	*B C A* *C A C* *A C D* *C D D*
Partial overlapping code	*B C A* *A C D* *D D A* *A B A*
Nonoverlapping code	*B C A* *C D D* *A B A* *B D C*

FIG. 1.—The letters *A*, *B*, *C*, and *D* stand for the four bases of the four common nucleotides. The top row of letters represents an imaginary sequence of them. In the codes illustrated here each set of three letters represents an amino acid. The diagram shows how the first four amino acids of a sequence are coded in the three classes of codes.

If each amino acid were coded by *two* bases (rather than the three shown in Fig. 1), we should only be able to code $4 \times 4 = 16$ amino acids. It is natural, therefore, to consider nonoverlapping codes in which *three* bases code each amino acid. This confronts us with two difficulties: (1) Since there are $4 \times 4 \times 4 = 64$ different triplets of four nucleotides, why are there not 64 kinds of amino acids? (2) In reading the code, how does one know how to choose the groups of three? This difficulty is illustrated in Figure 2. The second difficulty could be overcome by reading off from one end of the string of letters, but for reasons we shall explain later we consider an alternative method here.

. . . , *B C A*, *C D D*, *A B A*, *B D C*, . . .

or

. . . . *B*, *C A C*, *D D A*, *B A B*, *D C* . . .

FIG. 2.—The commas divide the string of letters into groups of three, each representing one amino acid. If the ends of the string of letters are not available, this can be done in more than one way, as illustrated. The problem is how to read the code if the commas are rubbed out, i.e., a comma-less code.

CODES WITHOUT COMMAS

BROAD
INSTITUTE

some make nonsense. We further assume that all possible sequences of the *amino acids* may occur (that is, can be coded) and that at every point in the string of letters one can only read "sense" in the correct way. This is illustrated in Figure 3. In other words, any two triplets which make sense can be put side by side, and yet the overlapping triplets so formed must always be nonsense.

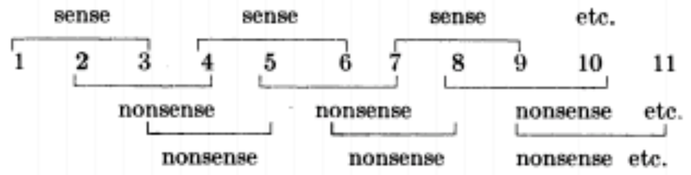

FIG. 3.—The numbers represent the positions occupied by the four letters A, B, C, and D. It is shown which triplets make sense and which nonsense.

It is obvious that with these restrictions one will be unable to code 64 different amino acids. The mathematical problem is to find the maximum number that can be coded. We shall show (1) that the maximum number cannot be greater than 20 and (2) that a solution for 20 can be given.

To prove the first point, we consider for the moment the restrictions imposed by placing each amino acid next to itself. Then, clearly, the triplet AAA must be nonsense, since, if it corresponded to an amino acid, α, then $\alpha\alpha$ would be $AAAAAA$, and this sequence can be misinterpreted by associating α with the second to fourth, or third to fifth, letters. We can thus reject AAA, BBB, CCC, and DDD.

It is easy to see that the 60 remaining triplets can be grouped into 20 sets of three, each set of three being cyclic permutations of one another. Consider as an example ABC and its cyclic permutations BCA and CAB. It is clear that we can choose any one of these, but not more than one. For suppose that we let BCA stand for the amino acid β; then $\beta\beta$ is $BCABCA$, and so CAB and ABC must, by our rules, be nonsense. Since we can choose at the most one triplet from each cyclic set, we cannot choose more than 20. No solution is possible, therefore, which codes more than 20 different amino acids.

```
"Not enough selections taking nucleotides 2 at a time to cover the
aminos but more than enough taken 3 at a time. Call them triples."

(def triples (map string (for [x ACGT y ACGT z ACGT] [x y z])))
triples          ; => ["AAA" "AAC" "AAG" "AAT" "ACA" "ACC" "ACG" "ACT"
;;                     "AGA" "AGC" "AGG" "AGT" "ATA" "ATC" "ATG" "ATT"
;;                     "CAA" "CAC" "CAG" "CAT" "CCA" "CCC" "CCG" "CCT"
;;                     "CGA" "CGC" "CGG" "CGT" "CTA" "CTC" "CTG" "CTT"
;;                     "GAA" "GAC" "GAG" "GAT" "GCA" "GCC" "GCG" "GCT"
;;                     "GGA" "GGC" "GGG" "GGT" "GTA" "GTC" "GTG" "GTT"
;;                     "TAA" "TAC" "TAG" "TAT" "TCA" "TCC" "TCG" "TCT"
;;                     "TGA" "TGC" "TGG" "TGT" "TTA" "TTC" "TTG" "TTT"]

(def rotations
  (fn [s] (let [n (count s)]
            (map string (take n (partition n 1 (cycle s)))))))

(rotations ACGT)                          ; => ("ACGT" "CGTA" "GTAC" "TACG")

(take 7 (map rotations triples))          ; => (("AAA" "AAA" "AAA")
                                          ;     ("AAC" "ACA" "CAA")
                                          ;     ("AAG" "AGA" "GAA")
                                          ;     ("AAT" "ATA" "TAA")
                                          ;     ("ACA" "CAA" "AAC")
                                          ;     ("ACC" "CCA" "CAC")
                                          ;     ("ACG" "CGA" "GAC"))

(set (str ACGT ACGT ACGT))                ; => #{\A \C \G \T}
[((set ACGT) \T) ((set ACGT) \Z)]         ; => [\T nil]

(def codons (set (map (comp set rotations) triples)))
(count codons)                            ; => 24
(take 4 codons)                           ; => (#{"ACC" "CCA" "CAC"}
                                          ;     #{"GGG"}
                                          ;     #{"TTT"}
                                          ;     #{"GCC" "CGC" "CCG"})

(map first (group-by count codons))       ; => (3 1)

(def sense-codons ((group-by count codons) 3))
(count sense-codons)                      ; => 20
```

Eureka!

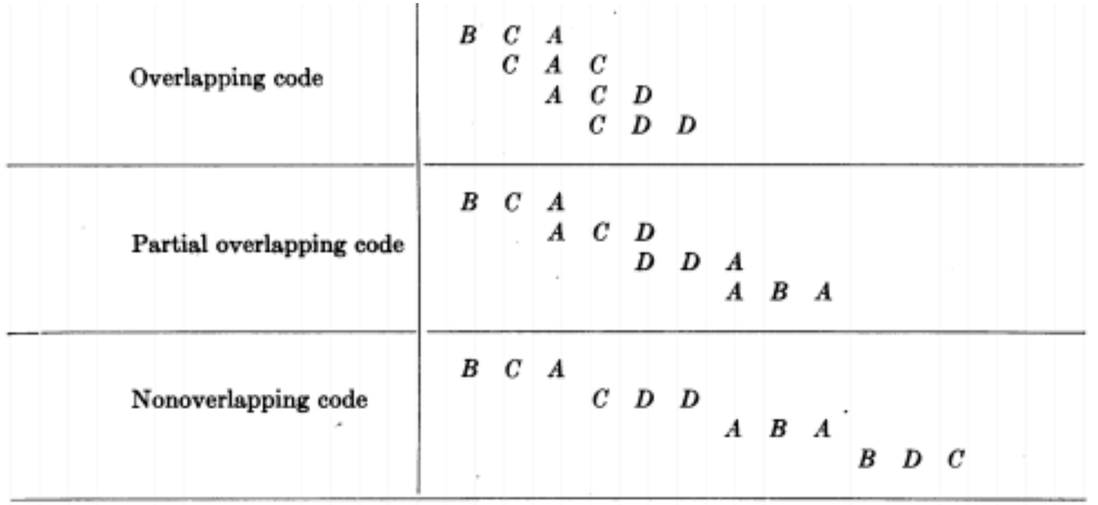

FIG. 1.—The letters *A*, *B*, *C*, and *D* stand for the four bases of the four common nucleotides. The top row of letters represents an imaginary sequence of them. In the codes illustrated here each set of three letters represents an amino acid. The diagram shows how the first four amino acids of a sequence are coded in the three classes of codes.

If each amino acid were coded by *two* bases (rather than the three shown in Fig. 1), we should only be able to code $4 \times 4 = 16$ amino acids. It is natural, therefore, to consider nonoverlapping codes in which *three* bases code each amino acid. This confronts us with two difficulties: (1) Since there are $4 \times 4 \times 4 = 64$ different triplets of four nucleotides, why are there not 64 kinds of amino acids? (2) In reading the code, how does one know how to choose the groups of three? This difficulty is illustrated in Figure 2. The second difficulty could be overcome by reading off from one end of the string of letters, but for reasons we shall explain later we consider an alternative method here.

. . . , B C A, C D D, A B A, B D C, . . .

or

. . . . B, C A C, D D A, B A B, D C . . .

FIG. 2.—The commas divide the string of letters into groups of three, each representing one amino acid. If the ends of the string of letters are not available, this can be done in more than one way, as illustrated. The problem is how to read the code if the commas are rubbed out, i.e., a comma-less code.

```
"Begin digression: Model base pairing with a simple Clojure map."

(str "Model" \space "base" \space "pairing" \.) ; => "Model base pairing."

(def sentence ["Model" \space "base" \space "pairing" \.])
sentence ; => ["Model" \space "base" \space "pairing" \.]

(apply str sentence)                            ; => "Model base pairing."
((partial apply str) sentence)                  ; => "Model base pairing."

(def string (partial apply str))
(string sentence)                               ; => "Model base pairing."

nucleotides            ; => ["Adenine" "Cytosine" "Guanine" "Thymine"]

(first nucleotides)                       ; => "Adenine"
(first (second nucleotides))              ; => \C
(map first nucleotides)                   ; => (\A \C \G \T)

(def ACGT (string (map first nucleotides)))
ACGT                                      ; => "ACGT"
(string (reverse ACGT))                   ; => "TGCA"

(def pair (zipmap ACGT (reverse ACGT)))
pair                                      ; => {\A \T \C \G \G \C \T \A}

(map pair ACGT)                           ; => (\T \G \C \A)

[(rand-nth ACGT) (rand-nth ACGT)]         ; => [\G \A]

(def strand (repeatedly (fn [] (rand-nth ACGT))))
(string (take 23 strand))                 ; => "TCTGTCCCCGTAGACAAGACGTT"
(string (take 23 (map pair strand)))      ; => "AGACAGGGGCATCTGTTCTGCAA"

"End digression: Now where were we?" "We were counting things."

(count ACGT)                                    ; =>  4
(count amino-acids)                             ; => 20
(count (for [x ACGT]                [x]))       ; =>  4
(count (for [x ACGT y ACGT]         [x y]))     ; => 16
(count (for [x ACGT y ACGT z ACGT] [x y z])) ; => 64
```