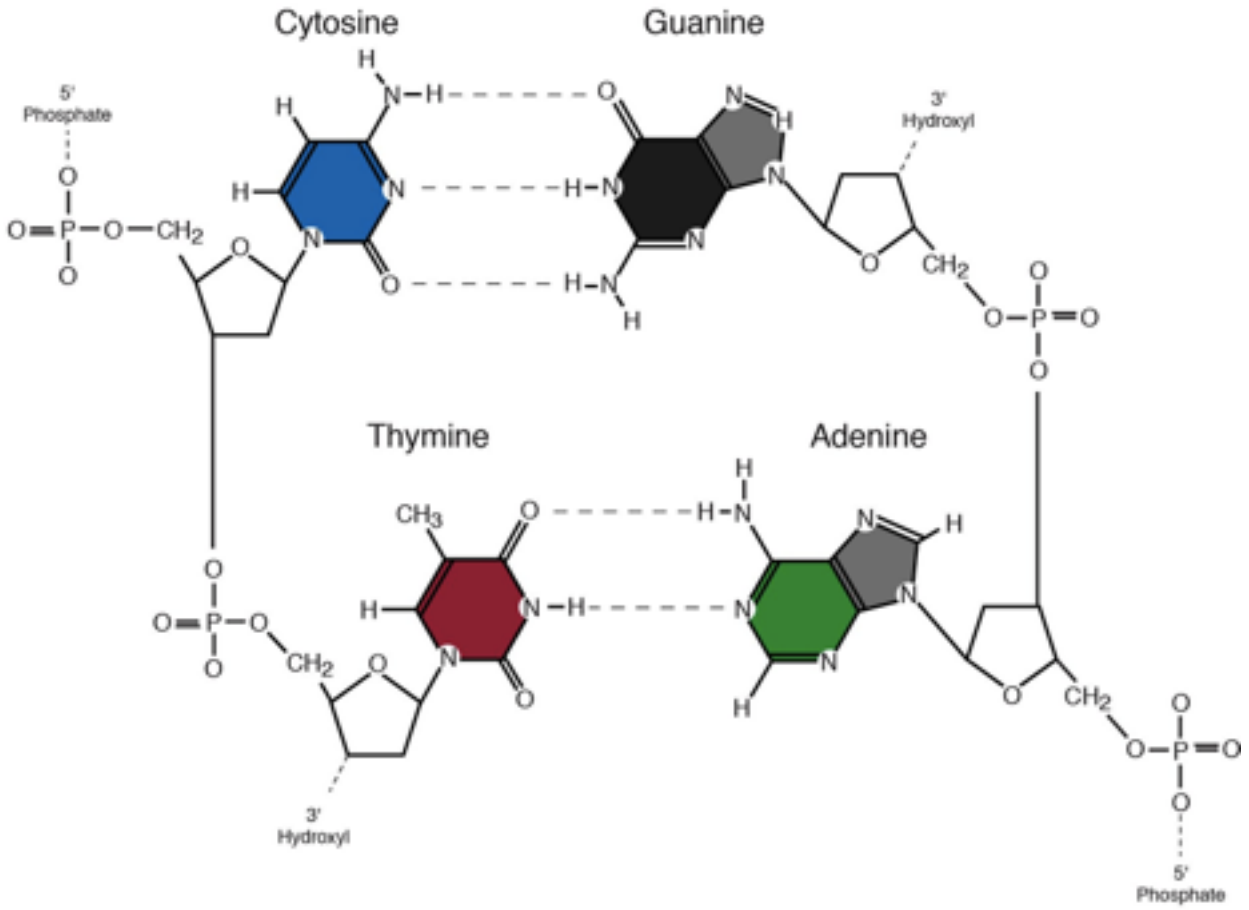
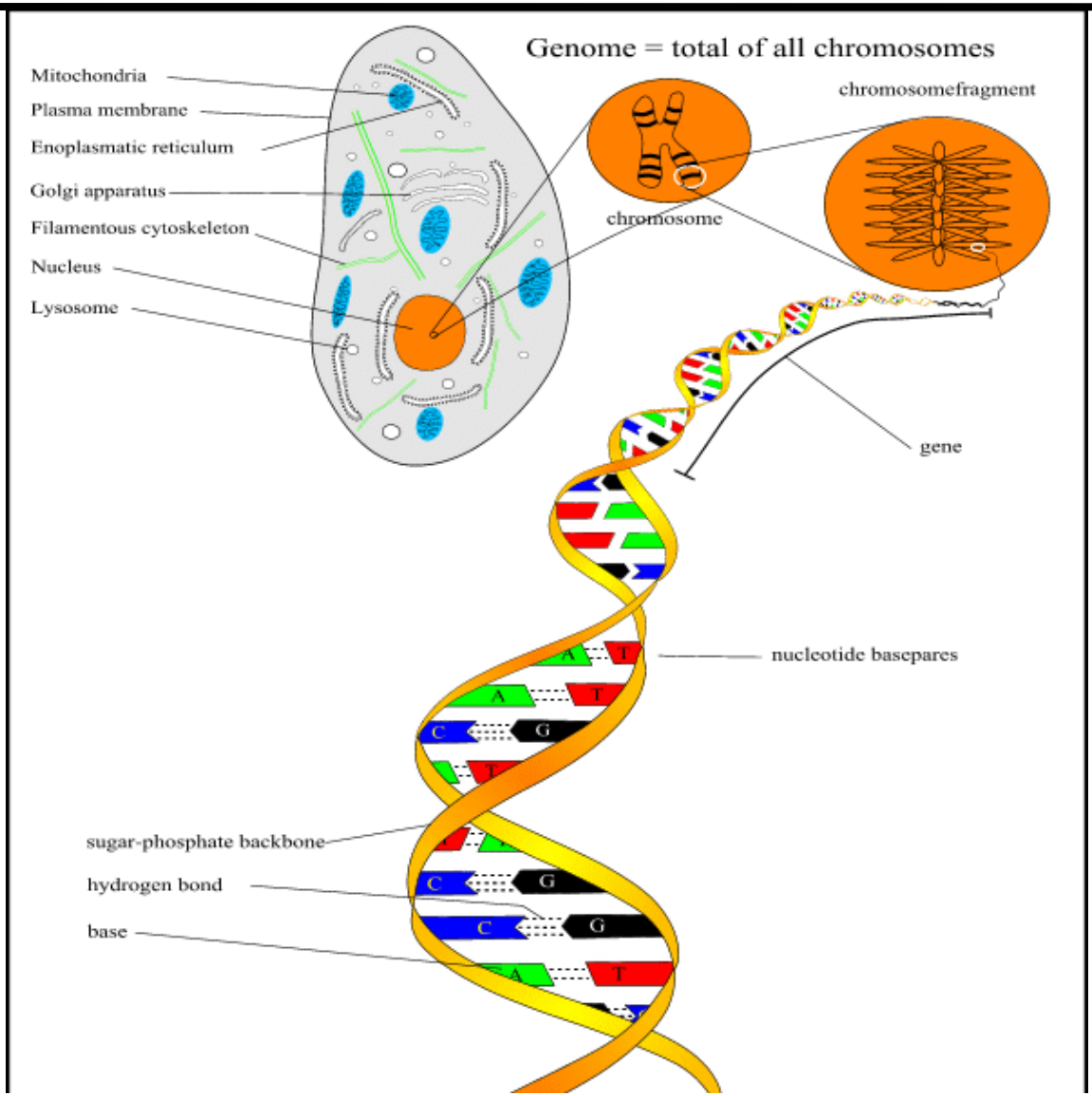
# Sequence Alignment

# Nucleotide and Base Pairs




- Purine: A and G
- Pyrimidine: T and C

# DNA


Genome = total of all chromosomes
Mitochondria
Plasma membrane
Enoplasmatic reticulum
Golgi apparatus
Filamentous cytoskeleton
Nucleus
Lysosome
chromosome
chromosomefragment
gene
nucleotide basepares
sugar-phosphate backbone
hydrogen bond
base
A
T
C
G

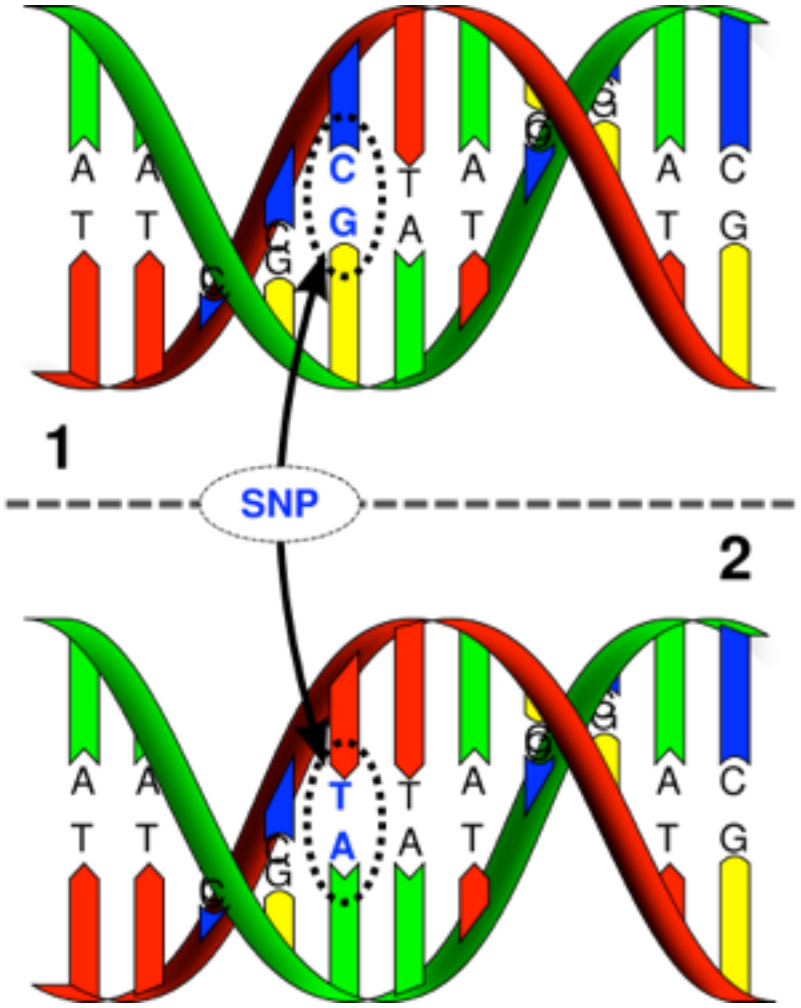
# For this course

- DNA is double-helical, with two complementary strands.
- Complementary bases:
  - Adenine (A) – Thymine (T)
  - Cytosine (C) – Guanine (G)
- What is the reverse complement of AAGGTAGC?

# DNA Mutation

- DNA mutates with a small probability when inherited by the offsprings.
- For example, one base can be substituted by another.
- This causes the differences between individuals of the same species.

# Single Nucleotide Polymorphisms




- Single base variation between members of a species.
- SNPs, which make up about 90% of all human genetic variation, occur every 100 to 300 bases along the 3-billion-base human genome.
- Major risk for genetic disease.

# Mutation Can Have Different Frequencies

- Two types of nucleotides:
  - Purine: A and G
  - Pyrimidine: T and C
- Two substitution types:
  - Transition: A ↔ G and T ↔ C
  - Transversion: otherwise.
  - Transition occurs more frequently. 2/3 of SNPs are transitions.

# Compare DNA sequences

- The most often used distance on strings in computer science is Hamming distance.
  - **AGTTTAATCA**
  - | | | | | | | | | |
  - **AGTATAACGA**
- This makes some sense on comparing DNA sequences in some cases. But there are other mutations
  - Substitution ACAGT → ACGGT
  - Insertion/deletion (indel) ACAGT → ACGT
  - Inversion ACA……GT → AG……ACT
  - Translocation AC……AG…TAA → AG…TC……AAA
  - Other genome level rearrangements.
- We only consider the first two mutations for now.
  - There are algorithms for the other mutations…

# Edit Distance

- E.g. CGATA and GGCCCATTA
- How “far” away they are from each other?
- The most commonly used distance is edit distance: the **minimum** number of edit operations it takes to convert one to another.
  - Edit operations often contain **substitutions** and **indels**
  - But there are extensions to this basic model.
- `d(ATGCATTTA,ATGTACTTTC)`
  - `ATGCATTTA`
  - `ATGTACTTTC`

# Edit distance is a distance metric

- Identity: d(x,y)=0 iff x=y
- Symmetry: d(x,y) = d(y,x)
- Triangular Inequality: d(x,z) <= d(x,y) + d(y,z)

# Alignment

- Hard to visually show the edit distance:
  - E.g. C→T@4, insert C@6, delete@9
- Alignment is much nicer:
  - ATGCA-TTTA
    |||  |  ||  |
    ATGTACTT-A
- Align the two sequences by inserting spaces, so that they are most **similar** column-wisely.
- What does “similar” mean?
- Usually we need a “scoring function” or a “score function”.
- E.g. Match = 1, mismatch =0, indel = 0.

# Alignment v.s. Edit Distance

- By a properly defined score scheme, finding the optimal alignment is equivalent to edit distance computing
  - match =
  - mismatch =
  - indel =

- Prove it!

# “Optimal” alignment

- The word “optimal” alignment is somewhat misleading. Ideally we want to find the “real” alignment of the sequences according to the real evolution instead.
- But we cannot!
- **This applies to most, if not all, bioinformatics problems. The “optimal” solution is not necessary the correct solution. It depends on how good the score function is**.
- The identity scoring scheme is not a very accurate one.
  - E.g., transitions and transversions have the same score.
  - Along this alignment topic, we will refine the score functions.

# Scoring sequence alignment

How to score an alignment?

Simplest scoring scheme:

- +1 = match
- −1 = mismatch
- −1 = indel

Let's see some examples

# Two sample alignments

```
AATGCGA-TTTT
  ||   | |||
G-TG--ACTTTC
```

```
AATG-CGATTTT
  || |   ||
G-TGAC-TTTC-
```

Which of the two alignments better?

# Alignment with DP

- The question is how alignment can be computed with a computer?
- Dynamic Programming
  - Requires a part of an optimal solution is also optimal.
- ```
  AATGCGA-TTTT
  |  ||   |  ||||
  A-TG--ACTTTT
  ```
- ```
  AATGCGA-TTTT
  |  ||   |  ||||
  A-TG--ACTTTT
  ```

# Last column of an alignment

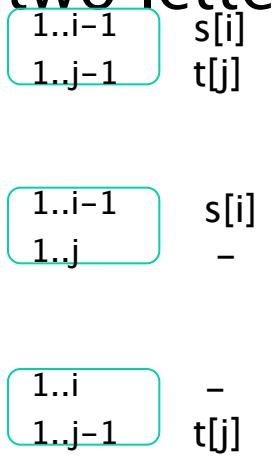
- Suppose we are to align S[1..i] and T[1..j]. Consider the last column of the optimal alignment. Three cases arise:

| | | | | | |
|---|---|---|---|---|---|
| S[1..i–1] | S[i] | S[1..i–1] | S[i] | S[1..i] | – |
| T[1..j–1] | T[j] | T[1..j] | – | T[1..j–1] | T[j] |
| (1) | | (2) | | (3) | |

- In each case, the sub-alignment without the last column is an optimal one (why?)

# Recursive definition

- Denote the optimal alignment score of S[1..i], T[1..j] by DP[i,j]. Then DP[m,n] is the optimal alignment score.
- Let f(*,*) be the score between two letters.
- Case 1: s[i] v.s. t[j]
  - DP[i,j] = DP[i−1, j−1] + f(s[i], t[j]);
- Case 2: s[i] v.s. −
  - DP[i,j] = DP[i−1, j] + f(s[i], −);
- Case 3: t[j] v.s. −
  - DP[i,j] = DP[i, j−1] + f(−, t[j]);
- Therefore…

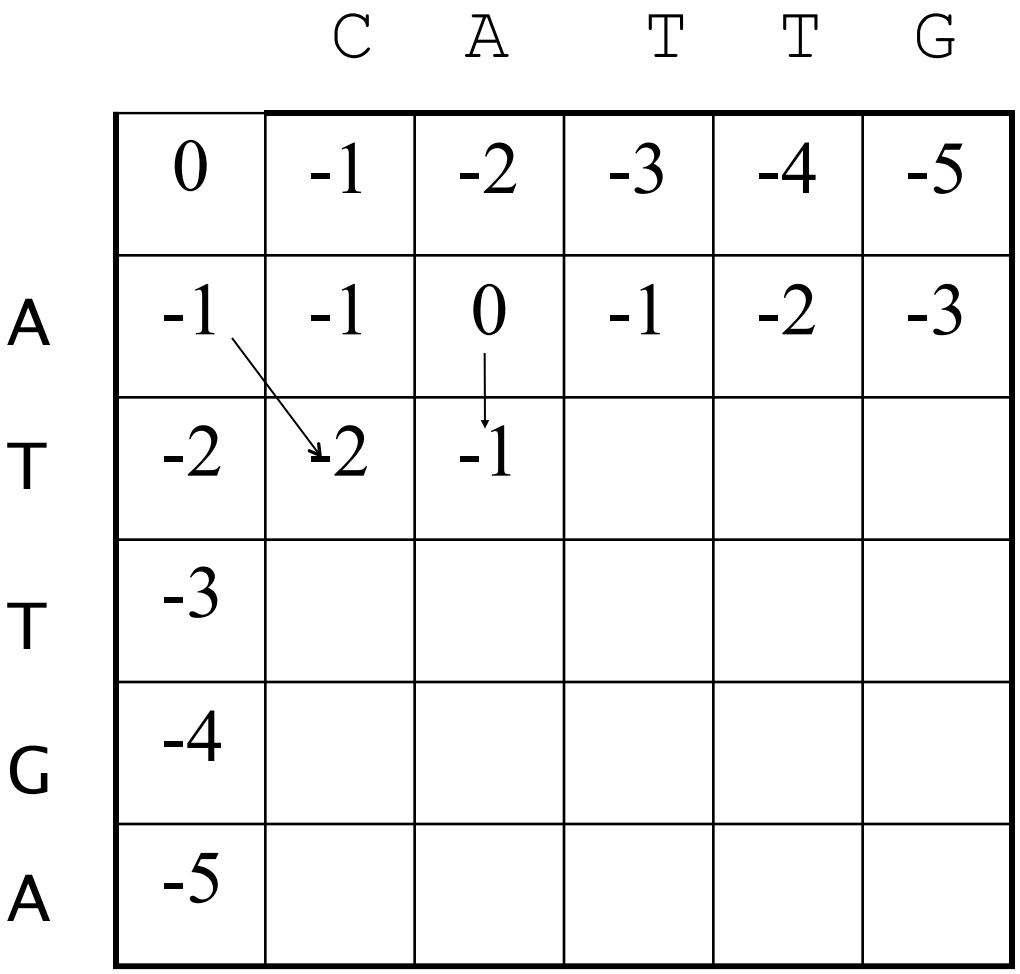



$$DP[i,j] = \max \begin{cases} DP[i-1, j-1] + f(s[i], t[j]); \\ DP[i-1, j] + f(s[i], -); \\ DP[i, j-1] + f(-, t[j]); \end{cases}$$

# Algorithm

```
DP[0,0] = 0;
for i from 1 to m
   DP[i,0] = i*  indel;
for j from 1 to n
   DP[0,j] = j* indel;
for i from 1 to m
   for j from 1 to n
                                  { DP[i-1, j-1] + f(s[i], t[j]);
      DP[i,j] = max               { DP[i-1, j] + f(s[i], -);
                                  { DP[i, j-1] + f(-, t[j]);
Output DP[m,n];
```

# Figure




CATTG-

-ATTGA

# Time Complexity

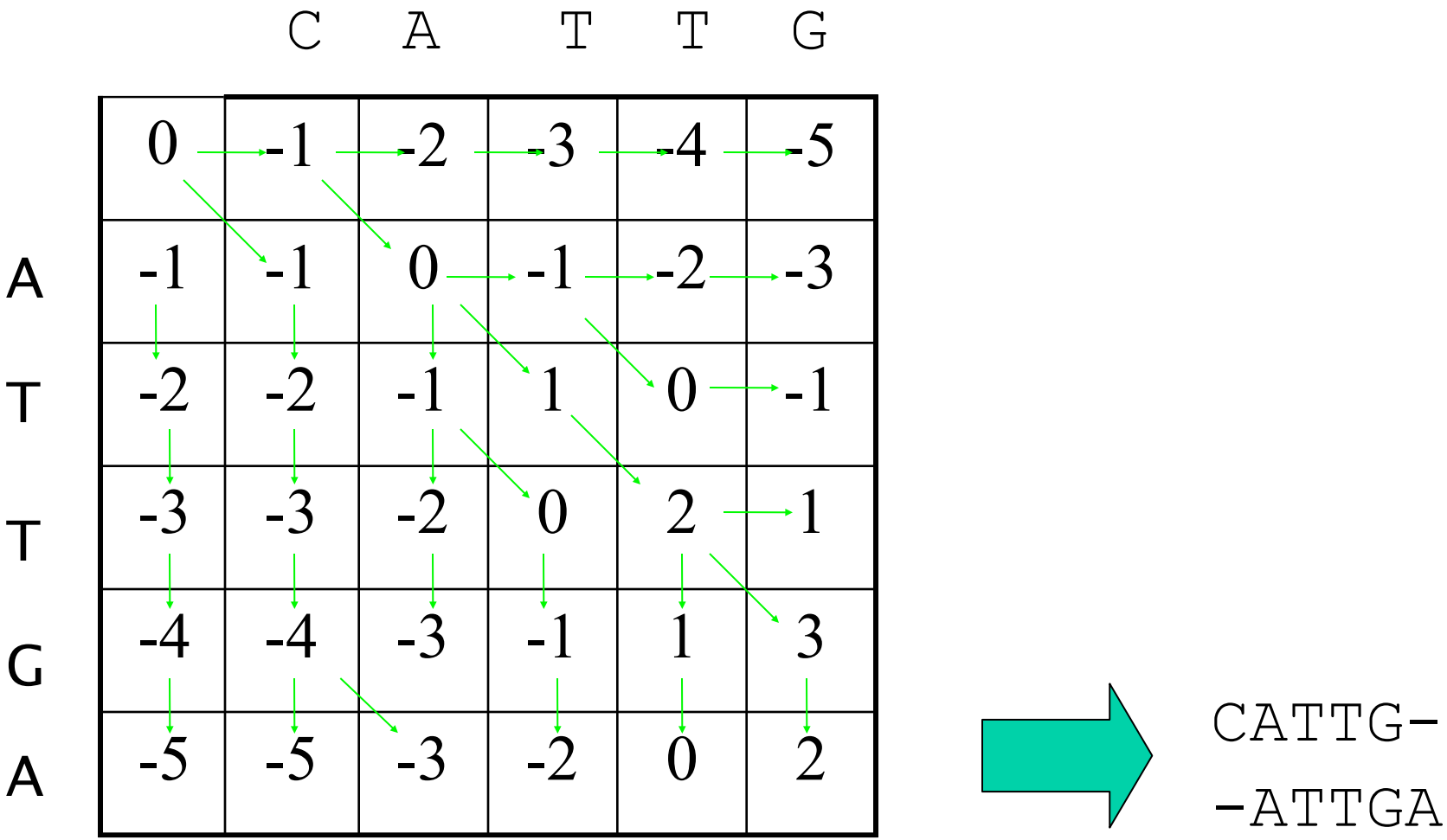
O(mn)

Question: What if write a recursive function?

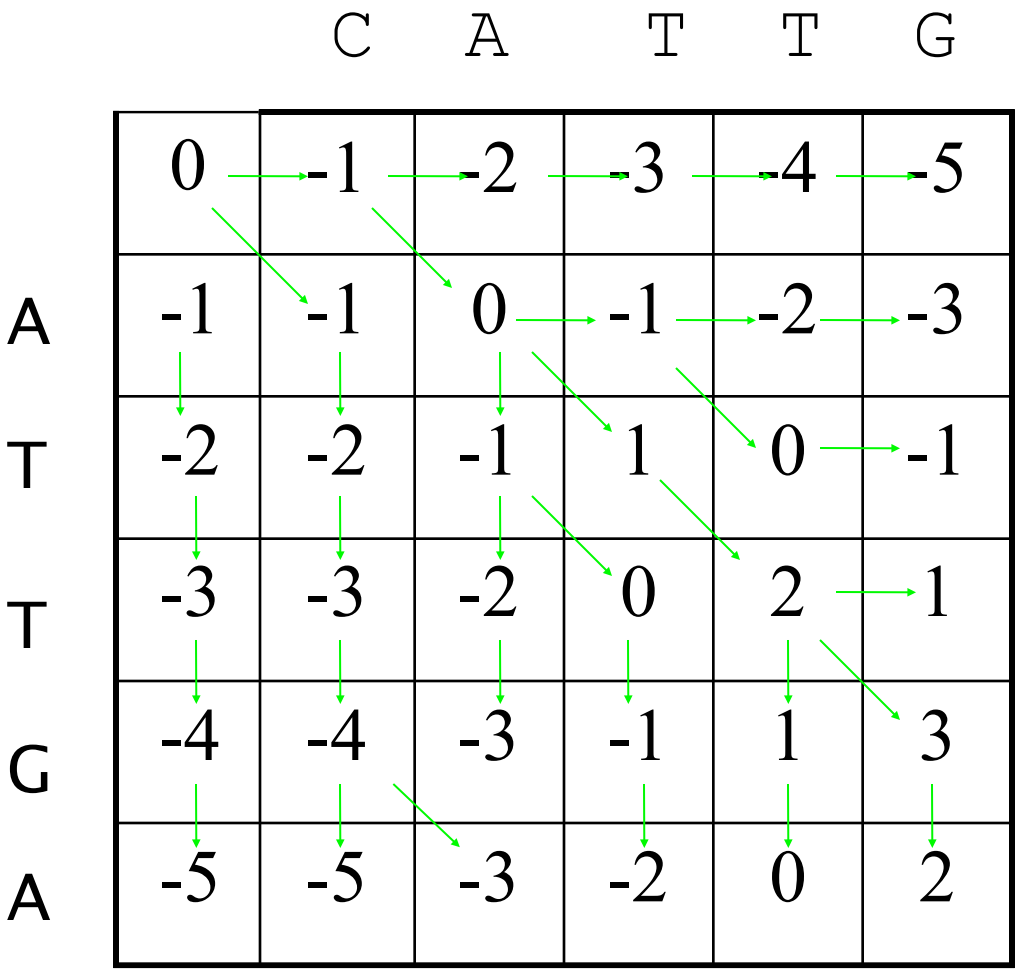
```
DP(int i,int j){
      x=DP(i-1, j-1) + f(s[i], t[j]);
      y=DP(i-1, j) + f(s[i], -);
      z=DP(i, j-1) + f(-, t[j]);
    return max(x,y,z);
}
```

What made the difference?

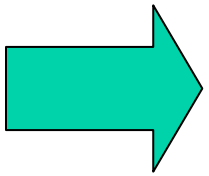
# Getting the actual alignment – backtracking



# Getting the actual alignment – backtracking




No need to physically record the green arrows. Why?



CATTG-
-ATTGA

# Pseudo Code for Backtracking

While i > 0 or j > 0:

- Figure out which of the three terms gave rise to M[i,j].
- Move to the right place (reduce i, reduce j, or reduce both), and write down the configuration of the current column.

```
S=T=empty string;
While i > 0 or j > 0
   if M[i,j]==M[i-1,j]+f(s[i],-)
      S = s[i]+S;
      T = '-'+ T;
       i--;
   else if M[i,j]==M[i,j-1]+f(-,t[j])
      S = '-'+S;
      T = t[j]+T;
      j--;
   else
      S = s[i]+S;
      T = t[j]+T;
       i--; j--;
```

Do you see the (potential) bugs in this program?

# Time Complexity

The dynamic programming algorithm runs in O(nm) time: Each step requires only 3 checks to other points in the matrix.

How about the backtracking?

# Space Complexity

We also need O(nm) space, to store the matrix.

If we only want to know the score of the optimal alignment, can you do better?

# Score Function

Now we have the algorithm for any score scheme f(.,.)

Next let's examine how to find a good score scheme.

Such separation of scoring and algorithm is a good thing.

> The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.
>
> – Dijkstra

# How to Build a Score Function

- First, know what you want.
- Purpose 1: alignment the same as true evolutionary history.
- Purpose 2: high score correlates to evolutionary correlation.
- We want purpose 1 if possible, but purpose 2 is also useful.

# Philosophy of a Score Function

- For purpose 1, right away: we might be **wrong**.
- That is, the alignment that has highest **score** may not be the one that actually matches evolutionary history.
- So you should never trust that an alignment must be right. It just optimizes the score.
- Should we give up purpose 1 at all?

# Philosophy of A Score Function

- For purpose 1, a scoring function may work some time, to some degree, in practice.
- As long as we know the limitation, we can still use it.
- For example, for the following alignment, it is "most-likely" the alignment is the same as the evolutionary history.
- `ACGTATTACCGT-TAACCG`
- `|||  ||||||||  ||||||`
- `ACGGATTACCGTATAACCG`
- Limitation we keep in mind: when score is low, alignment itself is not too useful.
- For purpose 2, we only aim to "approximate" the evolution distance.

# Improving the score

- Some simple improvements that do not affect the algorithm.
  - Give indel, mismatch, match different weights according to how rarely they occur during evolution.
  - Give different weights to transitions and transversions.
- Some other improvements affect the algorithm. E.g. gap penalty.

# Gap Penalty

```
AGATTTTTC      AGATTTTTC
AGA---TTC      AGA-T-T-C
```

Evolutionary history on the left seems "simpler" than the right.

→Indels are relatively rare. Indel a segment of $k$ consecutive bases is much easier than $k$ scattered indels.

→But linear gap penalty (k*indel for a length k gap) gives us no way of choosing between these two options.

# Arbitrary gap penalty

- Consecutive insertions or deletions are called a gap. Suppose the gap penalty of a length k gap is g(k) instead of the simple ck.
- Assume g(x)+g(y)<= g(x+y). (not necessary, but…)
- Can the old DP still work?

```
AATGCGA-TTTT
|  ||   |  ||||
A-TG--ACTTTT
```

score = 8+2*g(1)+g(2)

```
AATGCGA-TTTT
|  ||   |  ||||
A-TG--ACTTTT
```

This part is still optimal

```
AATGCGA-TTTT
|  ||   |  ||||
A-TG--ACTTTT
```

This part is not necessarily optimal

Let's try construct a counterexample.

# Arbitrary Gap Penalty

- We use D[i,j] to denote the optimal alignment score of s[1..i] and t[1..j]. definition
- D[i,j] = max of the following three cases:
  - D[i−1,j−1]+f(s[i],t[j]). (s[i] v.s. t[j])
  - $\max_{1 \le k \le i}$ D[i−k,j]+g(k)
  - $\max_{1 \le k \le j}$ D[i,j−k]+g(k)

algorithm

- Attention: when we do dynamic programming, the matrix itself has a meaning. Its definition is this meaning. The recurrence relation is the algorithm to compute it, but is not the definition.

# Time Complexity

- Cubic time complexity.
- In bioinformatics, the hardest work is perhaps choosing the right scoring function so that it both
  - approximates the real biology
  - can easily be computed
- Now let's simplify the g(k) a little. We basically want a function that grows slower than linear.
- g(k) = a + b*k
  - a = gap open penalty
  - b = gap extension penalty

# Affine Gap Penalty

With **affine gap costs**, the score of an alignment equals:

+1 for every match

−1 for every mismatch

−5 for every gap **open**

−1 for every gap **extension.**

(Of course, all of these can be arbitrary values, as we'll see later)

How to actually find the optimal alignment?

# Affine gap penalty example

- ATAGG--AAG
- || ||  | |
- ATTGGCAATG
- 6 match, 2 mismatch, 1 gap open, 2 gap extension, score = ?
- ATAGG-AA-G
- || || || |
- ATTGGCAATG

# Old Algorithm Does not Work

- Consider the last column of an alignment again:

```
AT-GG-     ATGG--
||  ||      ||  |
ATTGGC     ATTGGC
```

- When the last column is an indel, the added cost depends on the previous column.
  - If previous column has a gap opened already, then
  - D[4,6] = D[4,5] + gapext
  - Else
  - D[4,6] = D[4,5] + gapopen + gapext
- How do we know the previous column's configuration?
- Because by induction we know the optimal solution for D[i,j-1], can we simple look at it and use the configuration?

# How to solve the problem?

- Instead, we compute the optimal solution by forcing the last column to be one of the three configurations.

```
ATAGG      ATAGG-      ATAGGC
||  ||     ||  ||      ||  ||
ATTGG      ATTGGC      ATTGG-
```

$D_0[i,j]$ $D_1[i,j]$ $D_2[i,j]$

# Recurrence Relation

$$D_0[i,j] = f(s[i], t[j]) + \max \begin{cases} D_0[i-1, j-1]; \\ D_1[i-1, j-1]; \\ D_2[i-1, j-1]; \end{cases}$$

$$D_1[i,j] = \text{gapext} + \max \begin{cases} D_0[i, j-1] + \text{gapopen}; \\ D_1[i, j-1]; \\ D_2[i, j-1] + \text{gapopen}; \end{cases}$$

$$D_2[i,j] = \text{gapext} + \max \begin{cases} D_0[i-1, j] + \text{gapopen}; \\ D_1[i-1, j] + \text{gapopen}; \\ D_2[i-1, j]; \end{cases}$$

# Algorithm

- No difference to the simple DP but now uses three arrays.
- Backtracking should be very careful!
- Still O(nm) time. Approximately 3 times slower.
- This is okay because the model is more expressive.
- Much faster than the general gap penalty.

# Review: Evolution and alignment

Two sequences always arise from a common ancestor.

- Since that ancestor lived, there have been a long number of descendants, leading up to the present time.
- A full evolutionary history would detail the mutations that happened over the course of history.
- We don't have a time machine.

The next best thing: alignments.

Characterize which positions in the two sequences arose from the common ancestor.

Between these, "indel" mutations.

# Review

DP algorithm for alignment

- Matrix entry: score of best alignment of S(1…i) to T(1…j).
- Can compute matrix entries in constant time ➔O(nm) runtime.
- Can backtrack through matrix to find optimal alignment.
- If only score is needed, then linear space.

Scoring function important

- Some do not change DP (better scoring scheme)
- Some change (gap penalty)
- General gap penalty cubic time.
- Affined gap penalty still quadratic time.